Programming and Mapping Strategies for Embedded Computing Runtime Adaptability

Tiago Diogo Ribeiro de Carvalho

PhD

PRODEI: Doctoral Programme in Informatics Engineering

Department of Informatics Engineering (DEI), Engineering Faculty, University of Porto

Supervisor: Prof. Doutor João Manuel Paiva Cardoso

July 12, 2019
Programming and Mapping Strategies for Embedded Computing Runtime Adaptability

Tiago Diogo Ribeiro de Carvalho

Doctoral Programme in Informatics Engineering

FCT Fundação para a Ciência e a Tecnologia

Grant Number: SFRH/BD/90507/2012

July 12, 2019
Abstract

High-degrees of runtime adaptivity are expected in many future embedded computing systems. Runtime adaptivity foresees both adjusting the application and the mapping of computations according to usage contexts, operating environments, resources availability, etc. However, extending applications with adaptive features can be a complex task, especially due to the current lack of programming models and compiler support. Embedded computing systems include a vast myriad of computer architecture characteristics and execution environments, which makes more difficult for a compiler to statically map computations being aware of runtime changes and adaptations that may need to occur. In addition, the intermediate layer provided by virtual machines and JIT compilers makes even more complex to predict the impact of optimizations and further promotes runtime adaptivity and autotuning.

An option is to adapt applications using runtime mapping techniques that are aware of context usages and execution environments. The specification of runtime adaptivity is, however, a complex activity requiring different phases that include the identification of adaptivity opportunities, offline modifications and the inclusion of adaptivity code. These phases usually require different tools and source code modifications that may result in more complex applications, decrease readability, maintainability and consequently productivity. As each phase deals with one or more concerns, these concerns get scattered throughout the application code and/or are specified in different domain-specific languages and tool configurations.

This thesis focuses on schemes for programming strategies and compilation techniques for runtime adaptivity. It proposes an approach focused on programming runtime adaptivity strategies using the LARA domain-specific language, apart from the application logic, and the mapping of the computations by means of a source-to-source compiler leveraged by a runtime adaptivity API. The proposed approach helps code maintainability, experiments regarding the application of different adaptation strategies, and the generation of multiple implementations from the same application code when considering software product lines. An application, in early stages of development, takes advantage of the separation of concerns and the mechanisms for providing runtime exploration. An already developed application may take advantage not only from the separation of concerns and the runtime explorations but also from applying source-to-source transformations over the existing legacy code, an important feature when the code was not developed with adaptation in mind and code maintainability is required.

The approach to specify runtime adaptivity strategies is presented and evaluated in the context of Java programs, especially when considering runtime autotuning of optimization parameters and runtime selection of algorithms. We show the main advantages of the approach both in terms of the programming model and of the performance impact.
Resumo

Nos futuros sistemas embebidos de elevado desempenho é esperado um alto-nível de adaptabilidade em tempo de execução. Este tipo de adaptabilidade pretende ajustar uma determinada aplicação e mapear computações de acordo com os contextos de uso, os ambientes operacionais, recursos disponíveis, etc. No entanto, estender aplicações com características adaptativas é uma tarefa complexa, especialmente devido à falta de modelos de programação e de suporte por parte de um compilador. Os sistemas computacionais embebidos incluem uma grande variedade de características de arquitetura e ambientes de execução, tornando mais difícil para um compilador mapear estáticamente as computações tendo em conta as alterações e adaptações que podem ocorrer em tempo de execução. Para além disso, a camada intermédia das máquinas virtuais e compiladores JIT torna mais complexa a previsão do impacto de otimizações e promove ainda mais a adaptabilidade em tempo de execução e o uso de sistemas de afinação automática.

Uma opção é adaptar as aplicações através de técnicas de mapeamento em tempo de execução cientes do contexto e do ambiente operacional no qual estão a executar. A especificação de adaptabilidade de tempo de execução é, no entanto, uma atividade complexa que requer várias fases, que incluem a identificação de oportunidades de adaptabilidade, modificações offline e a inclusão do código de adaptabilidade. Essas fases geralmente exigem diferentes ferramentas e modificações no código fonte que podem resultar em aplicações mais complexas, diminuir a legibilidade, a capacidade de manutenção e, consequentemente, a produtividade. Como cada fase lida com uma ou mais requisitos, estes ficam espalhados por todo o código e/ou são especificadas em diferentes linguagens de programação e configurações de ferramentas específicas ao domínio.

Esta tese tem por objetivo o estudo de esquemas para a programação de estratégias e técnicas de compilação para adaptabilidade em tempo de execução. Foi desenvolvida uma abordagem focada na programação de estratégias de adaptabilidade de tempo de execução separada da lógica principal da aplicação e o mapeamento das computações por meio de uma ferramenta source-to-source apoiada por uma API com funcionalidades de adaptabilidade em tempo de execução. Esta abordagem ajuda à manutenção do código, na exploração da aplicação de diferentes estratégias de adaptação e na geração de múltiplas implementações a partir do mesmo código quando considerada as linhas de produção de software. Uma aplicação num estado inicial de desenvolvimento tem vantagens na separação dos requisitos e dos mecanismos para a exploração em tempo de execução. Uma aplicação já desenvolvida pode tirar proveito não só da separação de requisitos e da exploração em tempo de execução, como também da aplicação de transformações de código fonte quando aplicado a código legado, um recurso importante quando o código não foi desenvolvido com adaptação em mente e a manutenção do mesmo é necessária.

A tese apresenta e avalia o uso da abordagem para especificar estratégias de adaptabilidade em tempo de execução no contexto de programas Java, especialmente ao considerar o autotuning de parâmetros de otimização e a seleção de algoritmos. A tese também mostra os benefícios da abordagem em termos de modelo de programação e do impacto no desempenho.
Acknowledgments

This doctoral dissertation was one of the most difficult challenges I have endured and overcome, with many ups and downs that pushed me to the limit. It was a very enriching phase for me not only at an academic level but also at a personal level. Throughout this work, many people have helped me at different times and different situations and I am deeply grateful to all of them. Especially, my dearest thanks to my family, my closest friends and my work colleagues that, without them, I would not become who I am today.

On top of this group of people is my advisor, my teacher and my dearest friend João Cardoso, to whom I owe everything! Thank you for all the orientation and support given during this work and for being one of the most caring people I know. I am really happy I could work with you and proud to call you my advisor and my friend.

To my group colleagues, thank you for all the support and friendship during these years. To Ali Azarian, Nuno Paulino and Ricardo Nobre for your friendship and a special thanks to João Bispo, Pedro Pinto and Luis Reis not only for your friendship but for all your support and the continuous work on the development of our framework.

For always taking care of me and did everything so I could overcome any obstacle and become what I am today I thank my parents. Without all of your efforts and your love, I would not be here. A big thank you to my sister, the person that has supported me since I can remember. We had many fights since we were little, but we always watch each other’s back.

A special thanks goes to my girlfriend for all the unconditional love, support and patience during this work, and always cheering me up and giving me strength. It was hard but I did it and you are responsible for that! You convinced me to start, and helped, loved and cherished me throughout these years. You always stand by my side and for that, I am eternally grateful.

Finally, this thesis could not be done without the support given by the Fundação para a Ciência e a Tecnologia (FCT), under the PhD grant: SFRH/BD/90507/2012, and the research scholarships from Contexwa (POCI-01-0145-FEDER-016883) and SMILES (AE2017-0326 SMILES-5 - CSIG) projects.
# Contents

Abstract ........................................................................................................................................... i
Resumo ......................................................................................................................................... iii
Acknowledgments ......................................................................................................................... v
Contents ....................................................................................................................................... vii
List of Figures .............................................................................................................................. xi
List of Tables ................................................................................................................................. xix
Abbreviations ............................................................................................................................... xx

## 1 Introduction .......................................................................................................................... 1

1.1 Problem and Motivation .................................................................................................... 3
1.2 Objectives ......................................................................................................................... 4
1.3 Contributions .................................................................................................................... 5
1.4 Thesis Outline ................................................................................................................... 8

## 2 Background and Related Work ......................................................................................... 11

2.1 Aspect-Oriented Programming (AOP) ............................................................................. 13
2.2 The LARA Language ....................................................................................................... 15
   2.2.1 The LARA AOP Approach .................................................................................... 15
   2.2.2 Static Weaving ...................................................................................................... 19
2.3 Dynamic AOP ................................................................................................................... 21
   2.3.1 HotWave ............................................................................................................. 21
   2.3.2 PROSE ............................................................................................................... 23
   2.3.3 Steamloom .......................................................................................................... 26
   2.3.4 AspectWerkz ...................................................................................................... 27
   2.3.5 LLDSAL .............................................................................................................. 29
2.4 Runtime Optimization Frameworks .................................................................................. 31
   2.4.1 ADAPT .............................................................................................................. 31
   2.4.2 rePLay ............................................................................................................... 33
   2.4.3 SmartApps ......................................................................................................... 33
2.5 Algorithm Selection .......................................................................................................... 34
2.6 Domain-Specific Languages for Adaptation .................................................................... 34
   2.6.1 A DSL for Programming Runtime Adaptivity .................................................... 34
   2.6.2 DiSL ............................................................................................................... 36
   2.6.3 Elastic Computing .............................................................................................. 36

vii
7.6 Summary ................................................................. 161
8 Conclusions .............................................................. 163
  8.1 Concluding Remarks ............................................. 164
  8.2 Future Work ......................................................... 166
References ................................................................. 169
Appendix A ...................................................................... 179
Benchmarks .................................................................... 179
  A.1 Image Smooth Algorithm ....................................... 179
  A.2 FIR ................................................................. 181
  A.3 Matrix Multiplication ............................................ 181
  A.4 Weighted-Average Smooth ..................................... 182
  A.5 Median Smooth .................................................... 182
  A.6 Median by Sort Algorithms .................................... 183
  A.7 Sobel ..................................................................... 185
Appendix B ...................................................................... 187
Preliminary Results ......................................................... 187
  B.1 Experimental Setup ............................................. 187
  B.2 Results ................................................................ 190
    B.2.1 FIR Algorithm ............................................ 190
    B.2.2 Matrix Multiplication .................................... 192
    B.2.3 Sobel .......................................................... 194
  B.3 Analysis ............................................................... 198
About the Author ............................................................ 201
  Short Bio ................................................................. 201
  PhD-Related Publications ......................................... 202
Book Chapter .................................................................. 202
Journal Publications ...................................................... 202
Conference/Workshop Publications .................................. 202
Oral Presentations without Publications ............................ 202
Posters .......................................................................... 202
  Other PhD-Related Publications .................................. 203
Journal Publications ...................................................... 203
Conference/Workshop Publications .................................. 203
Posters .......................................................................... 203
List of Figures

Figure 2.1. LARA use in the context of a static weaving ........................................... 16
Figure 2.2. The sections of a LARA aspect (a) and an example of different types of aspects (b). .......................................................................................................................... 17
Figure 2.3. The use of LARA in different compilation levels (source: (Cardoso et al., 2016)). .................................................................................................................................. 18
Figure 2.4. LARA compile-time weaving design flow, including the front-end compilation (larac) and the weaving process over a target application (weaver). ........................................ 20
Figure 2.5. (a) A join point model hierarchy example, used in a MATLAB weaver, and (b) the attributes available for each join point ........................................................................................................ 21
Figure 2.6. HotWave2 architecture (Source: (Würthinger et al., 2010)) ..................... 23
Figure 2.7. PROSE interface between aspects and the JVDMI (Source: (Popovici et al., 2002))....................................................................................................................................... 25
Figure 2.8. A simple example of LLDSAL being used inside a function. The LLDSAL code dynamically generates a function to determine the current instruction pointer (source: (Payer et al., 2012)) ................................................................. 30
Figure 2.9. LLDSAL compilation flow (source: (Payer et al., 2012)) ....................... 30
Figure 2.10. ADAPT framework during compile-time and runtime (Source: (Voss and Eigemann, 2001)) ...................................................................................................................... 32
Figure 2.11. Adaptive strategy redefining an image size, according to the vehicle speed (Source: (Santos et al., 2011)) ......................................................................................................... 35
Figure 2.12. A hybrid approach for automatic specialization (source: (Khan et al., 2008)). ............................................................................................................................................ 40
Figure 2.13. Diagram defining the TOSKANA-VM system (Source: (Engel and Freisleben, 2005)) ............................................................................................................................................ 42
Figure 3.1. A recipe for an adaptive algorithm selection strategy considering execution time as the metric to minimize ........................................................................................................ 50
Figure 3.2. Compilation flow of the proposed Runtime Adaptation concept ................ 57
List of Figures

Figure 3.3 LARA compilation flow. ................................................................. 60

Figure 3.4. Representations of the code generated by the Weaver Generator (Interface), the code developed by the weaver developer (Weaving Engine), and the code of the source-to-source compiler. ................................................................. 61

Figure 3.5. Example of a join point definition in a Language Specification: the join point method and its selectable join points, attributes and actions. ................................................. 62

Figure 3.6. Timing calls to functions which name matches fName using (a) native code and (b) a LARA API. ................................................................. 64

Figure 3.7 Equivalent select statements since function is an alias for method in the Kadabra Weaver. ................................................................. 65

Figure 3.8. A synthetic method consisted of three nested loops invoking a single method. ................................................................. 66

Figure 3.9. Adapting call to foo every 10 executions of method bar. ....................... 67

Figure 3.10. Call to foo is adapted with new versions. The definition of new sampling versions is concurrent to the execution. ................................................................. 67

Figure 3.11. Call to foo is adapted with new versions but measurements are done around the outer most loop. ........................................................................ 68

Figure 3.12. Examples of code transformation and injection to prepare a program for adaptation. ........................................................................ 69

Figure 3.13. A call to (a) a static method named mult that pertains to a class named MatrixUtils and (b) the adapted code to allow runtime adaptivity. ......................... 69

Figure 3.14. The usual flow of an application with runtime adaptivity. ....................... 70

Figure 3.15. Possible adaptation schemes with the overhead the adaptation may impose. 72

Figure 3.16. Example of a sequential adaptation that decides between running the best configuration or explore and sample configurations provided by a search scheme. A version is executed “LIMIT” times and the adapter updates the reference to the best configuration according to the sampling measurements. ................................................................. 73

Figure 3.17. Example of a concurrent adaptation. The actual program and the adapter communicate via concurrent channels, allowing both to be executed laterally. The “offer” method allows asynchronous insertion in the channel, while the getters are synchronized methods. ...... 74

Figure 4.1. Example of a strategy that tunes the number of neighbors of kNN based on runtime criteria. ........................................................................ 78

Figure 4.2. Loop tiling with a fixed block size of 32 .................................................. 79

Figure 4.3. A high-level strategy that exposes the block size of a tiled loop and dynamically redefines its value with half the size of L2 cache, obtained using an API that retrieves CPU information at runtime. ........................................................................ 79
Figure 4.4. Loop tiling combined with an adaptation strategy that searches for the best block size (from 8 to 2048) based on the execution time of the outermost loop. The search uses increments of powers of 2 to reduce the exploration size .................................................. 81

Figure 4.5. Search scheme for the best block size that starts with the value 128 and when the sampling results in improvement it continues the search with the direct neighbors. .................. 82

Figure 4.6. A strategy that extracts a functional interface based on a method signature and extracts a method invocation to a variable. The algorithm to invoke is then dynamically adapted. ........................................................................................................... 83

Figure 4.7. A strategy similar to the strategy in Figure 4.6 that uses a map instead of a static switch, mapping a predefined scenario to the corresponding algorithm .................................. 84

Figure 4.8. Compile-time generation of a new interface and a class containing the exploration control fields and a method performing the search scheme. ................................. 85

Figure 4.9. Strategy for runtime exploration of algorithms based on scenarios only known at runtime. This strategy uses the functional interface and exploration class from the strategy shown in Figure 4.8 ............................................................................................................. 86

Figure 4.10. Algorithm exploration with a map that associates a knobs exploration to an algorithm .................................................................................................................. 86

Figure 4.11. Implementing specialization in a program may be done: (a) using conditional branches to select the best version, or (b) using a generative methodology. .................. 89

Figure 4.12. Java bytecode instructions snippet of the smooth method related to the two innermost loops and the inner operation: (a) original bytecodes; (b) bytecode template. ....... 92

Figure 4.13. A strategy that specializes at runtime a target method based on a Bytecode template and some runtime data ....................................................................................... 93

Figure 4.14. A strategy that generates at runtime specialized versions of a method based on a Bytecode template and some runtime data ................................................................. 93

Figure 4.15 Execution time speedup obtained from an exhaustive exploration of all possible values for two block sizes of two tiled loops on a matrix multiplication algorithm. .......... 94

Figure 4.16 Execution time speedup obtained from configurations for two block sizes of two tiled loops on a matrix multiplication algorithm ......................................................... 96

Figure 4.17. An exploration with three knobs using steepest descent scheme. .............. 97

Figure 4.18. Simplified Autotuner flow ........................................................................ 98

Figure 4.19. A strategy building an autotuner that searches for each possible runtime scenario the best configuration of two target knobs that improves execution time. ............... 99

Figure 4.20. A snippet of the injected autotuner code to control the knobs configuration and obtain runtime feedback with the specified measurement. ........................................ 99

Figure 4.21. A strategy building an autotuner that explores four different algorithms: two simple algorithms, an algorithm with a knob and a generative algorithm. .................. 100
Figure 5.1. High-level hierarchical representation of the Kadabra API developed for interaction with the developed Java API. ................................................................. 110

Figure 5.2. LARA aspect implementing the strategy depicted in Figure 4.1 that tunes the number of neighbors (k) to consider on kNN, based on runtime criteria. ......................... 111

Figure 5.3. Loop tiling tuning strategy defined in LARA that samples block sizes from 32 to a limit given as input matrix height, in powers of 2........................................ 112

Figure 5.4. A strategy that extracts a method call to a field and replaces the original call with access to that field. Then, before the new call is executed, an algorithm is assigned to the field based on the first input argument. ......................................................... 113

Figure 5.5. Example of a dynamic method adaptation with a template that requires two parameters. .................................................................................................................. 113

Figure 5.6. Example of a dynamic selection of the algorithm to use. When an algorithm does not exist for a given scenario, a new class is dynamically generated with a template accepting one argument................................................................. 114

Figure 5.7. Example of an aspect that tiles a loop and dynamically searches for the best tile size by means of a ControlPoint with an exhaustive (default) exploration.................. 115

Figure 5.8. LARA strategy that performs algorithm exploration at runtime using multiple algorithms.................................................................................................................. 116

Figure 6.1. A Java implementation of a matrix multiplication operation. ......................... 118

Figure 6.2. Results of all combinations of loop interchange for a matrix 1024x1024 when running in a PC with an Intel® Quad Core™ i5 650 @ 3,20GHz CPU and 8 GB DDRAM... 119

Figure 6.3. Weaving-time strategy (a) that interchanges loops j and k and extracts a[i][k] expression (see Figure 6.1, line 9) to a scalar variable and the resulting weaved code (b)...... 120

Figure 6.4. A weaving-time strategy that tiles the two innermost loops, defining the tiling loops around the outermost loop. ................................................................. 120

Figure 6.5. Matrix multiplication method of Figure 6.1 after applying loop interchange (j→k), scalar replacement and loop tiling over the two inner loops. The changes made over the code are highlighted. ................................................................. 121

Figure 6.6. Tiling strategy that explores block sizes from 32 to matrix height, in powers of 2......................................................................................................................... 122

Figure 6.7. A snippet of the code that is injected around mult call for the exploration.... 122

Figure 6.8. Same tiling strategy as Figure 6.6 using ControlPoint instead. ..................... 123

Figure 6.9. Tiling strategy that explores block sizes starting from half of the L2 cache size. The strategy starts by sampling lower values for the block and searches upwards when performance degrades......................................................... 124

Figure 6.10. A snippet of the code that is injected before mult call. The CpuInfo.l2 used in the aspect was translated to the CpuInfo.getL2CacheSize() method invocation................. 125
Figure 6.11. Strategy using the ControlPoint API with the same criteria as in Figure 6.9. .......................................................................................................................................................... 125

Figure 6.12. Definition of the elements necessary for the concurrent execution. This is part of the code assembling phase. The thread is started, and terminated, in the main method, given by the join point $mainMethod. ........................................................................................................................................ 126

Figure 6.13. Inserting code to be executed by the program, i.e., acquire configurations and provide time measurements to the channel. ............................................................................................................................... 127

Figure 6.14. Adaptation code to be executed in a concurrent thread. ................................................................. 127

Figure 6.15. Strategy snippet to explore random block sizes. This code replaces the adaptation body of Figure 6.14. ............................................................................................................................... 128

Figure 6.16. The FIR method in Java code. ........................................................................................................ 129

Figure 6.17. A simple approach to dynamically generate a specialized version of FIR when the coefficients window changes. .................................................................................................................. 129

Figure 6.18. Partial code of a strategy that explores $fir$ versions with different unroll factors. The if conditions only contain a pseudo conditional expression for simplicity purposes. .......................................................................................................................................................... 130

Figure 6.19. A strategy that builds an autotuner for the $fir$ method that compares the original version with a specialized version generated with ‘specializedToWindow’ template. .................................................................................................................................................. 131

Figure 6.20. A strategy that builds an autotuner for $fir$ method that compares the original version with a specialized version generated with the ‘specializedToWindow’ template. ........................................................................................................................................ 131

Figure 6.21. Smooth operation using the median value of a $3\times3$ window.................................................. 132

Figure 6.22. A strategy that generates a new functional interface based on the signature of the method medianNeighbor and extracts calls to that method to a field with the type of the new functional interface.................................................................................................................................................. 133

Figure 6.23. Strategy to build an autotuner that manages explorations of sorting algorithms for the different sizes the input argument of medianNeighbor can take. The exploration uses execution time to evaluate the algorithms........................................................................................................................................ 133

Figure 6.24. Example of algorithm selection for a kernel of size 3. Each callout represents the start of a version sampling. The callout with ‘***’ represents the point of best algorithm decision, from which all executions with that kernel size use the selected version. The experiments were performed in a PC with an Intel® Quad Core™ i5 650 @ 3,20GHz CPU and 8 GB DDRAM. .......................................................................................................................................................... 134

Figure 6.25. Strategy to build an autotuner that explores two normal sorting algorithms and two adaptive algorithms............................................................................................................................................... 134

Figure 6.26. Strategy to build an autotuner that explores two generative algorithms...... 135
List of Figures

Figure 6.27. Strategy based on profiling information that dynamically selects, the best sorting algorithm and generates a specialized version, based on the number of elements to sort. .............................................................. 136

Figure 7.1. Speedups obtained for different data sizes, for each processing kernel, when using two autotuning strategies: specialization based on kernel and unroll factor exploration. 145

Figure 7.2. Overhead of specializing the sorting network vs generating a new, specialized, class. .......................................................... 151

Figure 7.3. Sorting algorithm exploration to sort an array of integers (range=[0,255]), using three versions: quicksort (QS), counting sort (CS) and parallel counting sort (CS# followed by the number of threads). Parallel counting sort uses a local minimum search starting with 4 threads. The vertical dotted lines mark the locations where the explored algorithm is switched and the best algorithm found is marked within a bubble. The array was initialized with random values before each sorting iteration. .................................................. 152

Figure 7.4. Sorting algorithm exploration using the same exploration as in Figure 7.3 and considering the sorting of a collection of Person, ordered by age. ............................................. 154

Figure A.1. A weighted sliding-average smooth algorithm, using a WX by WY window. 179

Figure A.2. Example of a specialized version of the smooth algorithm, for a 3×3 window. .................................................................................. 180

Figure A.3. Speedups achieved for the median smooth algorithm with three specialization approaches: full loop unroll (LU), LU with constant propagation (LU+CP), LU+CP with data reuse (LU+CP+DR). ................................................................. 181

Figure B.4. The FIR method in Java code........................................ 190

Figure B.5. Specialized versions of FIR, for a window of size 4 (coefficients: {2,4,4,2}). ........................................................................ 191

Figure B.6. Speedups achieved with the specialized version of FIR, in two different target systems, for different window sizes. .......................................................... 192

Figure B.7. A Java implementation of a matrix multiplication operation........................................ 193

Figure B.8. Speedups achieved for the matrix multiplication in an x64 PC, with (right) and without (left) loop tiling. Matrices sizes of NxN. ...................................................... 193

Figure B.9. Speedups achieved for the matrix multiplication in Odroid, with (right) and without (left) loop tiling. ......................................................... 194

Figure B.10. Excerpt of the Java version of the Sobel algorithm............................................. 194

Figure B.11. Excerpt of the Java version of the convolution2d method. .............................. 195

Figure B.12. Code excerpt from a specialized version of sobel and the horizontal convolution................................................................. 196

Figure B.13. Speedups achieved with code specialization for the Sobel algorithm for an image size of 1024×768 pixels. ................................................................. 197
List of Figures

Figure B.14. Data reuse in the Sobel algorithm and its implications for the vertical and horizontal convolution operations after constant propagation and code elimination.............. 197

Figure B.15. Speedups achieved for Sobel without JIT optimizations. ......................... 198

Figure B.8.16. Speedups achieved (left axis) and the overhead applied (right axis), for different number of input images, when specializing the Median smooth (for the three tested windows) and Sobel programs. .......................................................... 200
List of Figures
List of Tables

Table 2-I. Comparison of the state-of-art approaches regarding DSLs for adaptation and Dynamic AOP................................................................. 44
Table 2-II. Comparison of the state-of-art Runtime Optimization Frameworks and approaches for Algorithm Selection......................................................... 45
Table 2-III. Comparison of the state-of-art: template-based code generation and tools for runtime adaptivity............................................................. 46
Table 2-IV. Comparison of the most significant approaches with the main characteristics intended in this work......................................................... 47
Table 3-I. DSL + API solution for the gathered user stories................................................. 58
Table 4-I. Description of the types of algorithms one may use in an algorithm selection strategy........................................................................................................ 87
Table 4-II. Example of code optimizations used for program specialization......................... 91
Table 5-I. List of some of the available join points and the join points that can be selected from those.................................................................................................. 108
Table 5-II. List of accessible attributes of some of the join points......................................... 108
Table 5-III Actions available per join point........................................................................... 109
Table 7-I. Strategies and the corresponding types of runtime adaptation applied. “n.a.” is used when adaptivity is not applicable in that strategy, while “✓” depicts the use of an adaptivity type.......................................................................................................................... 139
Table 7-II. Description of the metrics extracted from the Java source code.......................... 140
Table 7-III. Description of the metrics extracted from the code of the strategies................. 140
Table 7-IV. Description of the metrics extracted from the weaving process......................... 141
Table 7-V. Strategies’ results for 2048×2048 shaped matrices, including the total execution time and the adaptation overhead..................................................................... 142
Table 7-VI. Average execution time and speedup of matrix multiplication against Strategy 1-LoopInterchange when multiplying matrices of size 2048, with loop tiling over two loops and the same block size assigned to both tiles......................................................... 143
List of Tables

Table 7-VII. Results of executing the matrix multiplication with Strategies 1,3 and 4 over matrices of size 2048 by 2048, describing the execution time, the speedup against the version without loop tiling, and the number of explored version required to find the best version. The range given for the blocks is between 8 and 2048................................................................................................. 144

Table 7-VIII. Results of the overhead of each strategy, including the average, the minimum and the maximum time. ........................................................................................................................................ 144

Table 7-IX. The algorithm selected by the autotuner for each kernel size, between the original FIR (O), a specialized, generated, FIR (S), or an unrolled version with a specific factor (U [<factor>] ). ............................................................................................................................................... 146

Table 7-X. Average overhead (%) of the autotuning strategies, for each kernel size. The greener the cell is, the lower the overhead was ..................................................................................................................................... 146

Table 7-XI. Results of executing MedianSmooth with the defined strategies. Total time of Original version: 10 144s ............................................................................................................................................. 148

Table 7-XII. Average MedianSmooth execution time and speedup of the developed strategies ....................................................................................................................................................... 150

Table 7-XIII. Average adaptation overhead, in microseconds, of each strategy for each execution of method smooth. .................................................................................................................................... 151

Table 7-XIV - Java code before and after applying the different strategies ..................................... 155

Table 7-XV – Strategies wrote in LARA .............................................................................................. 155

Table 7-XVI – Strategies executed with the Kadabra weaver ................................................................ 156

Table 7-XVII – AO metrics about the changes made over the source code with the written strategies ........................................................................................................................................... 157

Table 7-XVIII - Java code before and after applying the different strategies ..................................... 157

Table 7-XIX – Strategies wrote in LARA ............................................................................................ 158

Table 7-XX – Strategies executed with Kadabra weaver .................................................................... 158

Table 7-XXI – AO metrics about the changes made over the source code with the written strategies ........................................................................................................................................... 159

Table 7-XXII - Java code before and after applying the different strategies over MedianSmooth and Integers and Objects sorting use cases .................................................................................. 159

Table 7-XXIII – Strategies wrote in LARA ......................................................................................... 160

Table 7-XXIV – Strategies executed with Kadabra weaver ................................................................. 160

Table 7-XXV – AO metrics about the changes made over the source code with the written strategies ........................................................................................................................................... 161

Table B-I. Characteristics of the Benchmarks used in the experiments .................................................. 188

Table B-II. Selected specialization experiments for each algorithm ..................................................... 189
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOP</td>
<td>Aspect-Oriented Programming</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CCC</td>
<td>crosscutting concerns</td>
</tr>
<tr>
<td>DBT</td>
<td>Dynamic Binary Translation</td>
</tr>
<tr>
<td>DSE</td>
<td>Design-Space Exploration</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain-Specific Language</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse Response</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
<tr>
<td>GPS</td>
<td>General-Purpose System</td>
</tr>
<tr>
<td>IPC</td>
<td>Inter-Process Communication</td>
</tr>
<tr>
<td>JIT</td>
<td>Just-In-Time (Compiler)</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>JVMDI</td>
<td>Java Virtual Machine Debug Interface</td>
</tr>
<tr>
<td>LLVM</td>
<td>Low-Level Virtual Machine</td>
</tr>
<tr>
<td>NFR</td>
<td>Non-Functional Requirements</td>
</tr>
<tr>
<td>OO</td>
<td>Object-Oriented Programming</td>
</tr>
<tr>
<td>RVM</td>
<td>Research Virtual Machine</td>
</tr>
<tr>
<td>TIB</td>
<td>Type Information Block</td>
</tr>
<tr>
<td>USN</td>
<td>Unfolded Sorting Network</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
</tbody>
</table>
1 Introduction

Contents

1.1 Problem and Motivation ................................................................. 3
1.2 Objectives ....................................................................................... 4
1.3 Contributions .................................................................................. 5
1.4 Thesis Outline .................................................................................. 8
A general-purpose system (Heath, 2002), such as a personal computer, can perform many different tasks as it is able to run software for different purposes. On the other hand, embedded systems (Gajski et al., 1994) are typically designed for specific functions, combining software and hardware components with either fixed or programmable capabilities. An embedded system is usually a component of a larger system (e.g., the brake system in an electronic car system), possibly controlling a specific functionality of the larger system. A general-purpose system is usually composed of several embedded systems, such as graphic cards or embedded controllers in keyboards. Some other examples of embedded systems are common daily devices, such as cellphones and house appliances (Heath, 2002).

An embedded system is not designed to be programmed in the same mode as a general-purpose system. Instead, the embedded system is built according to its specific functions. General-purpose systems can be used and programmed by the end-user with almost no restrictions in a general-purpose machine. An embedded system usually provides lower performance and, at the same time, have higher limitations on power and energy consumptions compared to general-purpose systems, however with the downside of being more difficult to program and restricted to particular uses (Heath, 2002).

The interest in embedded systems has increased significantly, being present in many of the daily devices, such as cellphones and car appliances. Research advancements (Compton and Hauck, 2002) provided important evolution in the area and, presently, it is possible to have high-performance embedded systems with the capabilities and performance sometimes close to the general-purpose systems (Wolf, 2014).

Embedded computing systems include a vast myriad of computer architecture characteristics and execution environments. This makes more difficult for a compiler to statically map computations being aware of the runtime changes and adaptations that may need to occur. An option is to research runtime mapping techniques aware of context usages and execution environments (Santos et al., 2011, Kavanagh et al., 2019). Very common is the fact that the modifications needed to adapt applications to certain usage contexts expose many mapping opportunities (Fleischmann et al., 1998). A core of a runtime adaptivity system (Daniel et al., 2008, Gadioli et al., 2015) is the specification of the runtime strategies and their execution during the execution of the application. These strategies have the possibility to access information only attainable at runtime and thus can feedback runtime data to compiler techniques, e.g., for code specialization (Voss and Eigemann, 2001).

The inclusion of adaptation strategies, and other types of secondary concerns, embedded in the application code, usually makes applications more complex, decrease readability, maintainability and consequently productivity. These concerns get usually scattered throughout the application code. By defining runtime adaptivity strategies apart from the application logic, developers can maintain a cleaner version of the original application, while it is easier to try different adaptation strategies, allowing a more maintainable and cleaner approach for both application logic and adaptation strategies (Sousa et al., 2004). Furthermore, this separation of concerns may make easier the generation of multiple implementations from the same input code (Czarnecki and Eisenecker, 2000).

The work presented here focuses on using a Domain-Specific Language (DSL) (Fowler, 2010), following an Aspect-Oriented Programming (AOP) (Kiczales et al., 1997) approach with dynamic weaving, to provide a novel runtime adaptivity methodology, in the context of embedded
systems. The proposed approach includes the use of software templates to expose and model adaptivity usage contexts and extensions to the LARA language (Cardoso, Carvalho, Coutinho, et al., 2012, Cardoso et al., 2016) to specialize such templates. These templates expose a higher abstraction level and a number of reconfigurability properties. The research of advanced forms of reachability analysis to identify the impact of certain runtime changes is an important aspect of this work.

1.1 Problem and Motivation

Advanced embedded applications, especially the ones in highly dynamic environments, may have to adapt to changes in contextual information (e.g., user’s location and activity (Ignatov, 2018, Wang et al., 2019)), or to changes in resource availability (e.g., energy (Flinn and Satyanarayanan, 2004) and connectivity) (Mukhija and Glinz, 2005). Adaptivity may face changes in application parameters (Epifani et al., 2009, Santos, 2014), application functionalities (Rinard, 2011), selection among different algorithms (Rice, 1976, Kotthoff, 2016, Wagner et al., 2018) (e.g., differing in computational complexity), different compiler optimizations (Voss and Eigemann, 2001, Meyer et al., 2018), hardware/software partitioning schemes (Stitt et al., 2003), options to map computations to reconfigurable units (Vahid et al., 2008), management of system resources (Mariani et al., 2010, Tsoutsouras et al., 2018) (e.g., switching and/or deactivating sensors), etc.

There are many applications where usage contexts leverage aggressive program specializations. Previous experiments with offline program specialization strategies to usage contexts revealed important hardware, software, and mixed (hardware/software) speedups (Coutinho, Carvalho, Durand, Cardoso, et al., 2012). However, when dealing with dynamic environments, optimizations and program specializations may provide higher positive impact if applied at runtime. Runtime strategies can adapt applications to the resources in the target computing system and may control resource allocation and task mapping to the cores of a multicore system, subject to specific requirements (Stitt et al., 2003, Marianik et al., 2011). Particular cases may highlight optimization opportunities such as data specialization (Knoblock and Ruf, 1996).

An example of possible runtime adaptivity is the use of specialized code according to data workloads and environments. Traditional approaches use multiple code versions generated offline by a compiler and at runtime, an adaptivity strategy is responsible to select among the code versions (Diniz and Rinard, 1997). However, the offline generation of multiple versions may increase program size, considering that to fully address a specialization problem, many versions might be required. Furthermore, in order to avoid code explosion, some specializations may have to be limited to a few values.

Compile-time specialization has its advantages, as it contains more time available to perform code generation and does not influence the code execution. However, the only information available at this time is static and/or speculative information. The proposed approach intends to follow this specialization but at the runtime level, by generating the request version with the given runtime information. Moving compiler optimizations and code generation to runtime can achieve important improvements (Kistler, 1997). However, this may impose unacceptable overhead and in order to keep the overhead low the compiler analysis and optimizations might be limited to specific and low-overhead optimizations (e.g., as with Just-In-Time (JIT) compilers (Aycock,
Introduction

2003). Hence, dynamic code generation requires more planning than an offline generation, considering that the online generation may affect program performance in less predictable ways. Using high-level source code generation may make generation easier, but requires more compilation steps, and thus requires more overhead time since the code has to be compiled before executed.

The current state of the art does lack a DSL for specifying runtime adaptivity strategies in the three most common dimensions: (a) configuration of parameters; (b) switch among algorithms; (c) mapping and optimizing according to each particular usage context. Furthermore, an integrated, unified and cross-layer view possibly provided by a DSL, tool and runtime support would make the specification and implementation of strategies for adaptivity (also considering the possibility to generate specific and specialized code on the fly) much easier.

1.2 Objectives

The main objective of this thesis is the research of novel techniques to both program and map runtime adaptivity strategies, in the context of advanced embedded computing systems (e.g., provided by state-of-the-art multicore processors). An important aspect of this work is the research of a programming model for specifying strategies, focused on extending typical application functionality with adaptivity.

The approach in this thesis is based on the belief that the specification of adaptation strategies shall be exposed to developers as a high-level DSL. This promotes the portability of the strategies, both in terms of target language and system and the exploration of different strategies in early development cycles. It is also believed that an approach promoting the separation of concerns (Parnas, 1972) is beneficial as it makes easier the debug, verification, optimizations, and the mapping of the behavior responsible for the runtime adaptation to specific computing cores. Furthermore, separation of concerns may require minor changes to the original application (i.e., the application without considering runtime adaptivity). A DSL, based on an AOP approach, allows the user to have a language focused on modifying a target application, without changing the source code, addressing from code monitoring (Cardoso, Carvalho, Coutinho, et al., 2012) to code specialization or runtime adaptivity (Voss and Eigemann, 2001, Santos et al., 2011). The DSL can provide low-level constructions allowing the rigorous definition of the modifications a strategy shall apply and where they shall take place. Moreover, it may also provide high-level constructs in which several low-level constructions may be invoked with user-defined heuristics.

The programming model shall allow the specification of strategies providing adaptivity at three different levels:

a) Adaptivity at parameter tuning;

b) Adaptivity at algorithmic choice;

c) Adaptivity at the code generation level.

The holistic approach to be addressed intends to allow the exploration of inter-dependences between these three levels. For instance, adaptivity to usage contexts commonly allows program specialization. The approach focuses on runtime adaptivity that includes mechanisms for runtime instrumentation, used, for instance, for profiling or debugging. The approach allows the developer
to try different strategies over the same application code, and even reuse some of the strategies in
different applications. The approach allows the specification of strategies for early development
cycles, with support for runtime versioning, profiling, and the development of adaptive strategies
based on that profiling. The code specialization considers the use of possible multiple pre-
compiled versions or the use of a low-overhead template-based approach, which generates
specialized versions at runtime, based on runtime information, an important step in order to avoid
the pre-generation of a great number of code versions and for cases where pre-generation is
impracticable.

Henceforth, this thesis is mainly focused on the following hypothesis:

“It is possible to improve computational metrics, related to programming
productivity, energy consumption, and execution time, by adapting an application at
runtime, using an aspect-oriented programming approach that leverages contextual
information and is orthogonal to the application logic.”

1.3 Contributions

The work presented throughout this thesis contributes on the study and analysis of the existing
work related to runtime adaptivity, on the research of the proposed approach and on the
development of a framework delivering the proposed features.

This thesis contributes with a compilation on the state of art in terms of dynamic AOP and
runtime adaptivity and optimization. An analysis and comparison between the existing work are
made in order to guide the engineering of the proposed approach. It also presents research and
collection of academics and industry user stories on the development of runtime adaptivity. The
user stores are analyzed and the requirements for the development of runtime adaptivity strategies
are gathered.

Based on these requirements and on the existing work, this thesis proposes an approach that
addresses the specification of strategies providing adaptivity at different three levels, namely at
parameter tuning, algorithmic choice, and code specialization level. The approach centers on
using a DSL with AOP concepts for the specification of the compile-time code restructuring and
the specification of runtime adaptivity strategies, combined with a weaving tool responsible to
apply the changes in the application and an API delivering runtime adaptivity mechanisms.

The approach allows the definition of different types of non-functional requirements, only
achievable at runtime, such as execution time reductions and energy savings. Furthermore,
following an AOP approach offers the developer more control on the application of secondary
corns (e.g., optimization of specific kernels with specific parameters), promoting, among other
features, program maintainability, program portability and developer productivity (Kiczales,
1996, Laddad, 2003). It proposes an approach that is orthogonal to some of the state of art
approaches, such as Santos et al. (2011) Voss and Eigemann (2001) and Ansel et al. (2009), with
ways to use these approaches in a systematic way.

This approach provides different types of runtime decisions for the developer, in order to
define where, when and how the strategy may take place. The where considers the selection of the
point of interest in the code, in which one intends to provide fine-grained access, including
Introduction

conditional expressions and local variables. These points of interest provide both static and runtime information and are the target of the required adaptation. The *when* considers the use of an event-driven approach, where the developer can decide if the adaptation occurs, for instance, every 50 milliseconds. The *how* considers the definition of the adaptation over the target point(s) of interest, considering a decision between the execution of sequential or parallel adaptation.

It presents the proposed methodology of the design of runtime adaptivity strategies, from the compile-time changes necessary to prepare the source code for the runtime adaptivity to the actual specification of the runtime adaptivity. The compile-time changes include restructuring the existing code and adding new features to the code. The specification of runtime adaptivity strategies contributes with three adaptivity types, providing adaptation for software and system parameters, the runtime selection of algorithms and the runtime generation and/or specialization of algorithms.

This work contributes with an approach for autotuning an application at the three proposed adaptivity types in two different modes. In one mode the autotuner adapts the application with a specific configuration to a given runtime scenario when both scenario and configuration is known. In the second mode, the autotuner allows the exploration of different configurations for unknown a given scenario. For a set of parameters, the exploration consists of sampling different configurations for those parameters based on a search scheme. The exploration allows exploring different types of algorithms, including typical algorithms, algorithms with parameters associated with them and generative algorithms. The algorithms with associated parameters can have specific search schemes for the exploration of different configurations for those parameters. A generic algorithm can immediately generate a new version based on the target scenario or it can use a search scheme to generate different algorithm versions to be explored.

It also suggests some search schemes to use for the runtime adaptation exploration, including full exploration, profile-based values, steepest descent, and local minimum, and the methodology to specify the adaptation goals. An adaptation goal is defined with a type of measurement to be used to score configurations and some criteria that compares the scores of the explored configuration and decides which one achieves that goal. The runtime adaptation decisions and code generation to be executed may be executed sequentially or concurrently to the application execution.

Most efforts were devoted to the compile and runtime support for dynamic adaptivity considering the use of an extended version of LARA (Cardoso, Carvalho, Coutinho, et al., 2012, Coutinho, Carvalho, Durand, Cardoso, et al., 2012), an AOP language, able to target multiple languages and systems, to specify strategies. More specifically, it proposes enhancements to the LARA framework to the development of new weaving environments, to be easy to update and extend and to allow more aspect decomposition for more reusability, including:

- an interpreter for the LARA language (*larai*) able to deal with different weaving environments;
- a weaver generator to develop new weaving environments interfacing with *larai*;
- new mechanisms for aspect composition and API-based strategies for reusability;
- the integration with Java and other OO languages;
- the development and integration of a Java-to-Java compiler;
Introduction

- a flexible approach allowing to add new features to the weaving engine and develop
generic/abstract/reusable strategies.

Based on the proposed approach and the enhancements and extensions over the LARA
framework, this thesis contributes with a Java to Java approach for programming runtime
adaptivity using the LARA programming language and its framework, a Java-to-Java weaving
engine (named Kadabra) developed with the LARA Weaver Generator and a Java API with
runtime adaptivity features.

The Kadabra weaver is able to finely select points of interest in the code and has several
code transformations that allow code restructuring to prepare the code for the runtime adaptivity
features. The Java API with runtime adaptivity features is used for supporting runtime autotuning
and adaptivity actions in the context of the strategies specified in LARA. These features include
the access to runtime information (e.g. program and system information) and runtime monitors
(e.g. timers), some concurrency utilities, search schemes providers, a template-based code
generator and an autotuning manager. The template-based code generator is able to dynamically
replace the bytecodes of a method or to dynamically generate new methods. The autotuner
manager can explore and sample algorithms and/or software knobs using time/energy monitors.

In order to ease the integration of the Kadabra weaver and the Java API for runtime
adaptivity a LARA API, named Kadabra API, is also provided. This facilitates the use of the Java
API within LARA strategies. This type of integration delivers a flexible approach that allows one
to easily add new types of runtime exploration, or even use a different API for adaptivity if one
has access to it.

It proposes a set of runtime adaptivity strategies designed in LARA in the context of several
kernels, widely known in both research and industrial areas. These kernels are presented as case
studies and the strategies demonstrate the methodology on defining runtime adaptivity in different
circumstances.

It presents an evaluation of the proposed approach in the context of the case studies to which
adaptivity strategies were defined. This evaluation shows the improvements of runtime adaptivity
and demonstrates the low overhead added to the applications, mostly not affecting the execution.
It also provides an analysis of the benefits of writing aspects in the DSL.
1.4 Thesis Outline

This thesis is organized in 8 chapters and provides also the bibliographic references, a brief biography of the author and an appendix at the end. The chapters are organized as follows.

Chapter 1, the current chapter, gives an introduction to the thesis topic. It provides a research contextualization, the problem statement and the motivation for this thesis. Then, the objectives of this thesis are presented followed by the definition of the main contributions.

Chapter 2 describes relevant background topics and related work to the thesis’ topic. It starts by giving some background information about aspect-oriented programming, one of the main concepts used in this thesis, followed by an overview of the LARA language. Relevant related work is discussed in this chapter, mostly focused on dynamic code manipulation and runtime adaptivity frameworks. Pros and cons are discussed at the end of the chapter and a comparison is made over the most relevant characteristics of the studied approaches.

Chapter 3 describes and analyzes the requirements for the development of runtime adaptivity strategies and describes the concepts of the proposed approach. It analyzes a set of user stories concerning the definition of runtime adaptivity strategies. Based on the requirements drawn from that analysis, the thesis approach for mapping runtime adaptivity strategies is proposed, describing how a DSL, a weaving framework and a runtime adaptivity API can accomplish those requirements. Then, it discusses the improvements done in the LARA framework to support the definition of runtime adaptivity strategies, followed by a discussion of relevant topics on the design of those strategies. These topics include the definition of execution points, the description of the compile-time restructuring and the adaptation description phases and how to evaluate the adaptivity at runtime.

Chapter 4 shows how runtime adaptivity can be designed and how it is expected that the runtime adaptivity is injected in the final application. It describes each type of adaptivity one may use in a runtime adaptivity strategy, shows what possible compile-time restructuring might be necessary and how the adaptivity is defined as a strategy. Three types of adaptivity are discussed: software and optimization knobs, algorithm selection and runtime code specialization. The concept of search schemes is also discussed here as the schemes to use for knobs exploration and also for runtime code specialization versioning. An automated approach to deliver runtime adaptivity for knobs and algorithms, named autotuner, is then presented. It describes how an autotuner can be used to explore different configurations, using search schemes, and decide at runtime the best configuration for a given scenario, based on configurable goals.

Chapter 5 describes the Java API for runtime adaptivity and the developed Java-to-Java source code weaver and shows how the proposed approach is used for the development of runtime strategies. It is presented the Java-to-Java weaver functionalities, including the language specification, i.e., the points of interest and the attributes one can access in a target application, and the actions that can be applied on those points. Then, it presents the LARA API that connects the Java API with the Java-to-Java weaver by generating the necessary code to interface a target application with the Java API. Subsequently, it describes how to develop the different types of runtime adaptivity strategies using LARA and the provided API, including the definition of autotuning strategies.
Chapter 6 presents cases studies for which runtime adaptivity strategies were developed. For each case study, several strategies were designed, each one as an increment of the previous strategies. It presents strategies for a matrix multiplication algorithm, in which loop tiling is applied to the algorithm and different search schemes are used to search for a block size that provides the best execution time performance. Next, runtime specialization strategies are described for the FIR algorithm, using a template that specializes the code to a specific kernel and a template that unrolls the loop processing the kernel with a provided factor. It also includes autotuning strategies that explore different unroll factors. Then, a set of strategies for selecting the best sorting algorithm is presented, using the median smooth algorithm as the case study.

Chapter 7 shows the experimental results of applying the runtime adaptivity strategies over the case studies. It presents and analyzes the performance results for each case study when executing the case studies with the developed strategies and it analyzes the overhead added by the runtime adaptivity code. Then several metrics concerning the definition of the runtime adaptivity strategies and the weaving process are presented and analyzed to show if the proposed approach definition improve the developers’ productivity to add runtime adaptivity to their target applications.

Chapter 8 concludes this thesis with some concluding remarks and describes some future work opportunities.
Introduction
2

Background and Related Work

Contents

2.1 Aspect-Oriented Programming (AOP) ................................................................................. 13
2.2 The LARA Language .................................................................................................. 15
   2.2.1 The LARA AOP Approach .................................................................................. 15
   2.2.2 Static Weaving .................................................................................................. 19
2.3 Dynamic AOP .............................................................................................................. 21
   2.3.1 HotWave ............................................................................................................ 21
   2.3.2 PROSE ............................................................................................................... 23
   2.3.3 Steamloom ......................................................................................................... 26
   2.3.4 AspectWerkz ..................................................................................................... 27
   2.3.5 LLDSAL ............................................................................................................. 29
2.4 Runtime Optimization Frameworks ............................................................................. 31
   2.4.1 ADAPT ............................................................................................................. 31
   2.4.2 rePLay .............................................................................................................. 33
   2.4.3 SmartApps ......................................................................................................... 33
2.5 Algorithm Selection ..................................................................................................... 34
2.6 Domain Specific Languages for Adaptation ................................................................. 34
   2.6.1 A DSL for Programming Runtime Adaptivity ...................................................... 34
   2.6.2 DiSL .................................................................................................................. 36
   2.6.3 Elastic Computing ............................................................................................. 36
   2.6.4 Adaptive Algorithm Selection for Reduction Parallelization ............................ 37
   2.6.5 PetaBricks ......................................................................................................... 37
2.7 Template-Based Code Generation ................................................................................. 37
   2.7.1 Circuit Generators ............................................................................................. 38
   2.7.2 deGoal ............................................................................................................... 38
   2.7.3 HySpec .............................................................................................................. 39
### Background and Related Work

<table>
<thead>
<tr>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.8 Frameworks for Runtime Adaptivity</td>
<td>40</td>
</tr>
<tr>
<td>2.8.1 ASM</td>
<td>40</td>
</tr>
<tr>
<td>2.8.2 Dynamic adaptivity in LLVM</td>
<td>41</td>
</tr>
<tr>
<td>2.9 Overview</td>
<td>42</td>
</tr>
<tr>
<td>2.10 Summary</td>
<td>47</td>
</tr>
</tbody>
</table>
This chapter starts by presenting Aspect-Oriented Programming (AOP) (Kiczales, 1996), with a subsection for AOP metrics, followed by a section about LARA and its background, including interpretation, weaver generation and the use of aspect composition. It presents important related work followed by an overview of such approaches and how LARA can overcome some problems of these approaches with respect to runtime adaptivity).

Several approaches for adapting an application, with different objectives and methodologies, have been already studied. Some approaches, such as (Santos et al., 2011) and (Cardoso, Carvalho, Coutinho, et al., 2012), make use of a DSL for specifying the intended changes on the application, while other approaches define libraries with a general-purpose language, such as Java and C++, for targeting applications, written in the same programming language (see, e.g., (Popovici, 2003., Marek et al., 2012)).

The AOP community also propose a number of approaches dealing with dynamic weaving, such as PROSE (Popovici, 2003.) and AspectWerkz (Bonér, 2004a), requiring (Villazón et al., 2009a) or not requiring (Bockisch et al., 2004, Nicoara et al., 2008b) modifications to the JVM. Independent from the type of definition, their weaving methodology changes according to the aim of their actions. While some change the source code at compile-time (Cardoso, Carvalho, Coutinho, et al., 2012), others comprise load-time weaving (Bonér, 2004a) or even runtime weaving (Würthinger et al., 2010).

Runtime adaptivity may be used for different objectives. There are approaches (Engel and Freisleben, 2006, Kuleshov, 2007) aiming to dynamically adapt an application for code instrumentation, while other approaches aim for dynamic program optimization (Patel and Lumetta, 2001, Rauchwerger and Amato, 2006, Charles et al., 2014), runtime specialization (Khan et al., 2008, Wernsing and Stitt, 2010, Lomüller and Charles, 2014) or even for runtime resource management (Mariani et al., 2010).

The following sections provide more state-of-the-art information regarding application adaptivity, for both compile and execution time, followed by a brief analysis and a summary of the studied approaches. It is analyzed how these approaches change the target applications, and how the adaptation strategies are defined (e.g., by a DSL).

### 2.1 Aspect-Oriented Programming (AOP)

Consider the example of a programmer who has worked extensively on a specific application domain and target platform. Over a period of time, this programmer would have acquired enough expertise to build a portfolio of strategies that allows him/her to satisfy given requirements, such as execution time, resource usage and energy efficiency. Such strategies typically involve applying complex code restructuring schemes, such as hardware/software partitioning, code specialization, source code transformations, or even insertion of monitoring modules to expose optimization opportunities at compile- or at runtime.

In a traditional development environment, this user knowledge is often captured by extensive manually modifications of the application code. As a result, this user knowledge and expertise (and other non-functional concerns in general) cannot be reused and applied systematically to either the same application, targeting different computing systems or across multiple applications. Furthermore, the resulting code becomes polluted with language constructs such as pragmas or...
Background and Related Work

conditional compilation directives, making it difficult to maintain and port, as different application domains may require different strategies.

These limitations on traditional programming languages imply the *tyranny of the dominant decomposition* (Tarr et al., 1999), allowing only the modularization of programs one way at a time. As a result, the rest of the “secondary” concerns that do not align with the modularization get scattered and scrambled throughout the modules of an application. These problems are normally called crosscutting concerns (CCCs) and are essentially concerns that do not align with the primary decomposition.

AOP (Kiczales, 1996) is a programming paradigm whose objective is to increase modularity by separating secondary concerns, the CCCs, from application’s core objective, resulting in cleaner code, easier concern analysis, easier for monitoring, tracing, debugging, etc. There are two types of symptoms related to the presence of CCCs: code tangling and code scattering (Filman et al., 2004). Code scattering can be seen when there are code fragments spread across the units of modularity, dealing with repeated code through the program. Code tangling can be observed in the crossing code modules, i.e., the primary concern appears tangled with code from other modules, becoming difficult to comprehend all the concerns in the unit (Filman et al., 2004).

Some fundamental concepts of AOP (Kiczales et al., 1997) are:

- **Crosscutting concerns**: the concerns to be modularized. These are the program secondary functionalities that are scattered or tangled throughout the program;
- **Join point**: a point of program execution. The execution point where to apply an aspect;
- **Pointcut**: a set of join points, usually in the form of a predicate that matches join points;
- **Advice**: an action to be taken in a join point. An Advice is related to a pointcut expression and acts over a join point related to the pointcut. This action can be applied before, after or around the specified join point;
- **Weaving**: the act of linking the aspects with the primary code, usually involving code injection on the selected points of program execution. When applying aspects into Object-Oriented Programming (OOP), a weaver links aspects with objects to make *advised* objects;
- **Join point shadow**: is a projection between the join point and the referenced point in the program code. In other words, it is the code programming element source of a join point.

Aspect components rely on three important artifacts: pointcut expression, advice, and weaver (Kiczales, 1996).

The pointcut expression expresses a set of join points. Each join point represents a program execution point. The advice performs actions over the join points. In turn, a static weaver applies the aspect modules over the application code/representation while a dynamic weaver will do the same but over the execution of the application.

There are several AOP languages, such as the ones described by Fabry et al in (Fabry et al., 2015). Those aspect languages try to aid programmers with mechanisms to achieve better modularity for their programs. Each AOP language addresses one or more purposes, being, for instance, AspectJ (Kiselev, 2002) and AspectC++ (Spinczyk et al., 2002) general-purpose AOP languages for Java and C++, respectively.
Most AOP approaches consider compile-time weaving, usually at source-code, outputting code with the concerns defined with the aspect language (Spinczyk et al., 2002, Aslam et al., 2010). There are also AOP approaches that do load-time or runtime weaving, for instance, over bytecode instructions, such as (Bonér, 2004a) and (Marek et al., 2012). The load-time weaving allows the advising of applications in binary form, usually by instrumenting when the application is loading modules (e.g., classes). Runtime weaving requires changes to the application during its execution by inserting directly code in the point of interest, or by halting the application, when that program execution point is reached. The last provides access to rapid prototyping since one can try different implementations without stopping, weaving and restarting the application (Bruneton et al., 2002, Popovici, 2003.).

Typically, AOP has been extensively used for inserting monitoring code and for modularization of cross-cutting functional code, and there have been many applications of the concept to parallel programming (Sobral et al., 2006), to configuration (Heuzeroth et al., 2001), and to specialization (Kaul and Gokhale, 2006). AOP approaches can be used for specifying general-purpose concerns including the ones related to application performance and taking into account the available resources and parameters. AOP can be also used to specify runtime adaptivity strategies.

### 2.2 The LARA Language

This section briefly describes LARA, an aspect-oriented language that directly addresses these development and maintainability issues (Cardoso, Carvalho, Coutinho, et al., 2012, Coutinho, Carvalho, Durand, Cardoso, et al., 2012). LARA allows developers to address non-functional requirements (NFRs) and to capture concerns in the form of strategies, completely decoupled from the functional description of the application. The strategies define which and in what order the application code can be modified and also allow developers to try different application designs (Coutinho, Carvalho, Durand, Cardoso, et al., 2012).

#### 2.2.1 The LARA AOP Approach

LARA (Cardoso, Carvalho, Coutinho, et al., 2012, Coutinho, Carvalho, Durand, Cardoso, et al., 2012) is a DSL centered on an aspect-oriented programming (AOP) approach (Kiczales et al., 1997), being the first versions mainly focused on offline changes (static weaving) to the application (Cardoso, Carvalho, Coutinho, et al., 2012, Cardoso, Teixeira, et al., 2012). A key element of LARA and a distinctive feature from existing approaches is the specification of strategies to satisfy NFRs and user knowledge in a non-invasive way as well as the support for exploration of alternative transformations and the mapping of alternative implementation with combined hardware/software co-design solutions. LARA was developed in the context of the FP7 EU-funded REFLECT project (Cardoso, Diniz, et al., 2011), having roots in AOP languages such as AspectJ (Kiczales et al., 2001) and AspectC++ (Spinczyk et al., 2002), and has also been inspired by previous work on extending MATLAB with aspect-oriented concepts (Cardoso et al., 2010).

Figure 2.1 shows the use of LARA aspects to weave, at compile-time, an application code. LARA is partially agnostic to the target language (Cardoso, Coutinho, et al., 2012), hence not tied to a particular language. The language is flexible enough to support extensions to different target
languages and weavers. The aspects are defined the same way for any input language, which facilitates the aspect definition if one desires to apply a similar aspect to different target languages, such as MATLAB and C (Cardoso, Carvalho, Coutinho, et al., 2012).

**Figure 2.1. LARA use in the context of a static weaving.**

LARA includes both declarative and imperative semantics. The semantic of pointcut expressions and the associated advices is fully declarative, whereas the semantics of the code implemented inside functions and advice sections is governed by an imperative model. Moreover, LARA is intended to describe strategies that affect design flows by conveying to the design stages of such flows, specific code transformations, compiler optimizations, or even target system properties.

Figure 2.2 (a) illustrates the structure of a LARA aspect. Aspects can be decomposed in different `aspectdef` blocks for better modularity and aspects can recursively invoke other aspects. Each `aspectdef` comprises a set of optional preliminary statements, such as input and output parameters, static fields, and conditional checks to test if an aspect can run with the given inputs. For the main scope, LARA uses the JavaScript\(^1\) syntactic and semantic elements for the common code statements such as assignments, loops, conditional statements, among others; and weaving statements to select pointcuts and advise over the join points. The weaving statements, depicted in the gray code block of Figure 2.2 (a), consist of `select`, `apply` and `condition` statements, the essential statements in LARA to define strategies. The `select` statement defines the join point selection over the source code (Carvalho, 2011).

The `apply` statement can obtain information of each join point and advise over the set of join points related to a pointcut expression (in a `select` statement). To filter the join points, one can use the attributes filter in the select statement or use a `condition` statement over an apply statement (Carvalho, 2011).

Background and Related Work

LARA 2.0 (Cardoso, Coutinho, et al., 2012) is the version of the LARA language developed in the context of the REFLECT project. It was designed to allow the development of hardware/software solutions that meet specific requirements. To this extent, LARA allows developers to (1) codify user knowledge and expertise regarding non-functional concerns, (2) apply and reuse compilation and synthesis strategies in a systematic fashion targeting multiple applications and platforms, and (3) allow aspect modular composition thus facilitating design space exploration (DSE). Figure 2.2 (b) depicts some examples of aspects aiming for different goals: an instrumentation aspect (InstLoops), a strategy to map computations (MapFunctions), and an outer aspect using both previous aspects (Main).

LARA includes various “operative” and transformational commands, recognized as weaving actions by LARA interpreters. The invocation of weaving actions in LARA enables programmers to describe specific aspects of the behavior of the weaving, thus allowing the definition of strategies that guide and control tools in a design-flow. An immediate application of this mechanism, for instance, includes the use of strategies to implement design patterns consisting of sequences of compiler optimizations. The use of LARA can be thought of as a way to convey complementary information and guiding strategies to assist a toolchain to map an application to the target architecture. Furthermore, the support of code instrumentation in LARA is essential to automate application code analysis, tuning and debugging.

Figure 2.3 shows the use of LARA to support developers in mapping their applications to embedded systems (including heterogeneous high-performance systems) across various design stages including code analysis, monitoring, mapping decisions, and hardware/software compiler optimizations. In addition to supporting traditional AOP mechanisms, the LARA approach focuses on mechanisms that address crosscutting concerns exposed by various design

---

2 One of the first versions of LARA has been presented in (Cardoso, Carvalho, Coutinho, et al., 2012).
Background and Related Work

elements, such as the application source code, tools used in the design flow, and the target platforms.

LARA allows developers to capture application NFRs in a structured way, leveraging high/low-level actions and flexible toolchain interfaces. Developers can thus benefit from retaining the original application source code while also leveraging the benefits of an automatic approach for various domain-specific and target component-specific compilation/synthesis tools. Specifically, LARA has been designed to help developers reach efficient implementations with low programming effort.

In essence, LARA uses AOP mechanisms to offer in a unified framework: (i) a vehicle for conveying application-specific requirements that cannot otherwise be specified in the original programming language; (ii) the use of requirements to guide transformations and mapping choices, thus facilitating design space exploration (DSE); and (iii) an extensible interface for the various compilation/synthesis components of the toolchain. As illustrated in Figure 2.3, LARA aspects are used to describe concerns (possibly crosscutting) using different abstraction levels, from the specification of the concerns to strategies based on actions to implement those concerns. This allows a unified view of concerns and the use of the same language (avoiding the need to learn more than one language) over different specific concretizations and design flow levels. Figure 2.3 also illustrates the different levels of abstraction using a specific crosscutting concern: “maximize performance on platform X”. This concern is specified and applied as LARA strategies. These LARA strategies may consist of monitoring, mapping and guiding, specializing, and DSE actions. The actions are then applied in a weaving stage that can be at the source-to-source level, at the compiler optimizations level, and/or at the DSE level.

Figure 2.3. The use of LARA in different compilation levels (source: (Cardoso et al., 2016)).
2.2.2 Static Weaving

An important concept of AOP is the notion of weaving (Kiczales et al., 1997). With LARA, the weaving process combines, in an automated fashion, functional and non-functional concerns leading to the desired implementation. There are several benefits to the weaving process as pursued by the LARA-guided design-flows:

- It allows non-functional concerns to be developed and maintained independently from the original application source code. This decoupling promotes a clean separation between the algorithmic specification and nonfunctional descriptions leading to a cleaner and thus easier to maintain source code base;

- LARA aspects can specify strategies that capture a set of transformation steps to achieve different NFRs thus leading to potentially distinct implementations. Aspects can be applied and updated on the basis of different types of requirements without directly affecting the original source code. This feature substantially improves overall portability and application code maintainability;

- Aspects can be developed independently from application source code and therefore reused in the context of multiple applications. This reuse of aspects allows non-expert developers to exploit specialized transformations and strategies geared toward specific target architectures, thus substantially promoting productivity and portability across similar target architectures;

- The ability to specify generic and parameterizable aspects in LARA is particularly useful for describing hardware- and software-based transformation patterns and templates, thus facilitating DSE. Examples of aspect parameters include application- and domain-specific information.

The LARA approach supports a generalized weaving process in which actions are tied to the program code base, to its intermediate representations or to the target system architecture. Another key innovation of this approach lies in bringing together, in the same framework, various types of transformations and operational aspects in the mapping of computations to embedded systems. LARA allows developers to specify different types of concerns, mainly addressed by the following concepts:

- **Monitoring**: Specification of monitoring features, such as the current value of a variable or generation of control flow graphs, providing insight for the refinement of other aspects;

- **Specializing**: Definition of specific code/properties for a particular input code, when targeting a specific system;

- **Mapping and Guiding**: Specification of mapping and compiler optimization strategies, which embody mapping actions to guide tools to perform specific implementation decisions (e.g., mapping arrays to specific memories, applying loop unrolling to loops meeting certain conditions);

The compile-time design of LARA is depicted in Figure 2.4. LARA aspects are processed by a LARA front-end component (larac, in Figure 2.4), which converts a LARA aspect file into an executable specification in XML called the aspect intermediate representation (Aspect-IR) (Coutinho, Carvalho, Durand, and Cardoso, 2012). This Aspect-IR is the integration data-structure used by LARA to link to the target weaver. A weaving tool is then responsible to parse
Background and Related Work

the target application and to interpret the Aspect-IR, in order to apply the required changes over
the target code, i.e., to weave the strategies in the target application. The output of the weaving
tool is a modified version of the application including the code resultant from the advices of the
aspects.

Figure 2.4. LARA compile-time weaving design flow, including the front-end compilation (larac) and the
weaving process over a target application (weaver).

The front-end requires three language specification files: (1) the join point model
representing the points of interest in the input programming language; (2) the join point attributes
defining properties associated with each join point type; and (3) the action model describing each
possible action that an aspect can perform on a join point. The language specification is both
associated with the target language and to the weaver. The join point model defines the code
structure in which
the weaver works and the points of interest it can
select and
advice. Figure
2.5(a) depicts a parcel of a join point hierarchy from a weaver that targets MATLAB applications.
The hierarchy allows one to choose explicitly, for example, all the loops available in the source
code, or to select implicitly one specific loop by its function parent or by its nested level (Cardoso,
Carvalho, Coutinho, et al., 2012, Bispo et al., 2013).

This join point selection can be filtered by the information the weaver obtains from each join
point. That information, i.e., join point attributes, allows a fine-grained selection over the join
points and monitoring information regarding the selected join point. Some of the attributes in a
MATLAB-related attribute model are illustrated in Figure 2.5(b). The model comprises the
fundamental join points present in MATLAB code (Cardoso, Carvalho, Coutinho, et al., 2012,
Bispo et al., 2013).
Background and Related Work

Figure 2.5. (a) A join point model hierarchy example, used in a MATLAB weaver, and (b) the attributes available for each join point.

An action in AOP is the act of changing the target program in the selected join point. Code insertion, before, after or around the join point, is a common action in the AOP languages. It is used for monitoring, profiling and tracing purposes. Generally, insertion is the predefined action in AOP languages (Kiczales et al., 1997). Actions in LARA are not a static part of the language since the application of actions is the responsibility of the weaver. The action model is used for defining the type actions a weaver can apply over a join point. For instance, a weaver that allows code insertion and the use of optimization tools, such as loop unrolling, can define its action model with these actions. Another example is the define action available, e.g., for MATLAB programming language which allows the definition of types of variables (Carvalho, 2011, Bispo et al., 2013).

2.3 Dynamic AOP

The approaches used by the AOP community for dealing with dynamic AOP and dynamic weaving (see, e.g., (Bockisch et al., 2004, Nicoara et al., 2008b, Villazón et al., 2009a)) provide useful background for the research presented in this thesis. The AOP community has proposed a number of possible dynamic weaving implementations, e.g. in the case of Java, requiring (Villazón et al., 2009a) or not requiring (Bockisch et al., 2004, Nicoara et al., 2008b) modifications to the JVM. The use of a virtual machine provides a layer between the virtual instructions and the processor instructions. These approaches are analyzed in the context of this thesis in the following sections.

2.3.1 HotWave

HotWave, HOTswap & reWeAVE (Villazón et al., 2009b), focus on runtime weaving over bytecode instructions. Based on AspectJ and compatible with standard JVM, it uses similar techniques as FERRARI, a framework for bytecode instrumentation (Binder et al., 2007). While the former allows static bytecode instrumentation over any method, HotWave has the ability of
Background and Related Work

dynamic instrumentation and code hotswapping. It relies on the class redefinition mechanism provided by the package java.lang.instrument. Aspects defined in HotWave can be woven and re-woven at runtime, in any method with a bytecode representation, including the standard Java library and classes generated at runtime (Villazón et al., 2009a).

An aspect in HotWave is defined in AspectJ language since the framework itself comprises both AspectJ compiler and weaver. HotWave provides the pointcut designators, such as execution, call, get, set, etc., and access to static and dynamic join point information, identical to AspectJ. The transformation is made in the weaving process, as HotWave process runtime instrumentation and does not support static crosscutting. Java’s hotswapping mechanism is the cornerstone for the runtime weaving or reweaving. It is used for weaving aspects to already loaded classes, while newly loaded classes are woven at load-time with the most recent aspect. Once an aspect is no longer required, HotWave allows dynamic unweaving, reverting a class to its original version (Villazón et al., 2009a).

In HotWave advices can be applied before and after the advising join point. For the around advice, HotWave does not provide a direct method. A communication mechanism is provided in turn, called inter-advice communication, which allows advices to share a local variable for passing information. The communication is accomplished with a static field, declared in the Aspect class with @InvocationLocal annotation. The declared static fields are woven in the target method as local variables, and the advices using these fields are mapped into the new defined local variables. Therefore, this mechanism can be used to emulate an around advice, which consists of wrapping a join point with before and after advices, using an invocation-local variable when data communication is required between the advices. This emulation provides a basic around advice for wrapping the join point, since repeating or replacing the join point is not conceivable. Another use of the inter-advice communication mechanism is the emulation of a pointcut control flow (cflow), similar to the cflow of AspectJ (Kiczales et al., 2001), which retrieves all join points within a join point control flow.

HotWave provides runtime adaptation that allows weaving previously loaded classes, a feature not available in many approaches which only provides load-time weaving, where any method with bytecode representation can be weaved. Its dynamic weaving is compatible with AspectJ and any standard JVM (Villazón et al., 2009a).

Since the class redefinition, using standard JVMs, is restricted, HotWave only provides modifications to a method body and the constant pool. Changing a class structure, e.g. adding new members or changing class hierarchy or a method signature are not conceivable. For these reasons, a new version of HotWave was developed, based on a new JVM that provides access to what was once a restriction in standard JVMs (Würthinger et al., 2010). The new JVM allows adding and removing members from a class, as well as changing super types and interfaces.

In the first version, HotWave supports static join points by adding new static fields, to keep the static join point information, to existing classes. However, it requires a hack in the ClassLoader, comprising the security model of Java. In HotWaver2, the static join points do not require this hack and allow the creation of static fields in any class. Also, as previously mentioned, HotWave does not support around advices. The new version provides this advice since it uses the capabilities of the new JVM to easily redefine classes and methods (Würthinger et al., 2010).

The architecture of HotWave2 is depicted in Figure 2.6. HotWave2 can use the AspectJ weaver and performs the bytecode transformations weaving. The new JVM, named JavaHotSpot
Virtual Machine, provides advanced hotswapping to redefine a class (Würthinger et al., 2010). The virtual machine (VM) can apply arbitrary changes during the program execution, in an already loaded class. In order to add a new aspect, HotWave2 bootstraps the VM using a standard Java agent to register the new class transformer. Reweaving a class can be triggered with the provided API, which always weaves the original class (Würthinger et al., 2010).

![HotWave2 architecture](source:image)

This new HotWave version provides higher access and flexibility over any Java application and can apply further changes over a class, or method than what is possible with a standard JVM. However, HotWave2 only works with the new JVM and no compatibility exists with standard versions (Würthinger et al., 2010).

### 2.3.2 PROSE

PROSE, PROgrammable extenSions of sErvices (Nicoara et al., 2008b), is a dynamic AOP approach addressing Java applications. The platform centers the use of dynamic aspects for fast prototyping and debugging for applications under development, preventing the halt of the application, redeployment, and reboot of the new prototype. The authors give more emphasis in programmability, in terms of aspects writing and insertion, rather than application performance. The system provides a Java library used to define the aspects in plain Java, which avoids the use of a new or an extended language and does not require third-party tools for weaving.

An aspect in PROSE consists of a Java class which extends the class Aspect, included in the PROSE library. Each aspect implementation contains a set of crosscut object declarations. These crosscut objects are similar to a pointcut plus the concerning advice in AspectJ (Kiselev, 2002). Therefore, each crosscut declaration contains a pool of join points and the corresponding advice to execute over those join points.

After the aspect(s) definition, its insertion can be done both by direct insertion on the JVM or by an extension manager provided by PROSE. The extension manager is executing parallel to the application execution. Once an aspect is inserted by means of the PROSE extension manager, such as:

```java
Aspect asp = new ExampleAspect();
Prose.extensionManager().insert(asp);
```

The extension manager searches for the intended join points and requests the JVM the application halt when these join points are reached (Popovici et al., 2002).
Background and Related Work

PROSE avoids the practice of Meta Object Protocol (Kiczales et al., 1991), a common approach of other platforms such as BCEL\(^3\). Since it was one of the first approaches to target dynamic aspects, PROSE went through three stages, each with different methods of join point capture and weaving process.

The first stage (Popovici et al., 2002) was based on using the Java Virtual Machine Debug Interface\(^4\) (JVMDI), which can retrieve runtime state and change the application execution in the JVM. It contains mechanisms to produce event notifications on the join points, making it possible to pause the application in the associated program execution point and execute the intended concern. After the advice execution, the application returns its natural execution. This process requires the execution in debug mode, stopping the application to execute, externally, the advice.

Figure 2.7 illustrates the aspects insertion (a) and the weaving process (b) once a join point is reached. PROSE uses an extension manager to interface the aspects with the JVMDI, called JVM Aspect Interface, and provide aspect insert at runtime. Once one aspect is inserted (1), PROSE core searches (2) for the join points invoked and requests notifications, from the debugger layer, to register the stop requests (3). When a join point is reached (b), the program execution is suspended and the debugger layer gives control to PROSE (4). The advice for the join point is invoked (5) and, after its execution, PROSE returns the program control (6), resuming its execution.

\(^3\) BCEL website: http://commons.apache.org/proper/commons-bcel/
\(^4\) JVMDI is deprecated since J2SE 5.0. Currently, JVM Tool Interface replaces JVMDI
With this JMVDI approach, it is not possible to add new members, such as fields and
methods, since this implementation does not support source-code or bytecode instrumentation.

On a second stage, PROSE gives the user the option to use the Just-In-Time (JIT) compiler
(Popovici et al., 2003). The option allows one to weave hooks in the source code to mark the
intended join points and execute the advices determined by the hook points.

The third stage (Nicouara et al., 2008a) is based on dynamic bytecode instrumentation. The
runtime adaptation is done with method code replacement (Nicouara et al., 2008a), which consists
of replacing the original method code. This methodology still avoids the creation of new classes
or methods. As an alternative, it inserts new functionality, at runtime, by recompiling the desired
methods to replace them with the advice, and inlines the proceed method (the join point actual
execution) in this advising code method. This last stage is the closest approach for achieving
better performance.

In summary, the PROSE platform is capable of weaving an application at runtime without
stopping its execution. The platform takes advantage of programming aspects in plain Java, using
the predefined PROSE library. It is suitable for applications in development for rapid prototyping or when dynamic reconfiguration according to environment changes (Nicoara et al., 2008b).

2.3.3 Steamloom

Steamloom (Haupt, 2005) is an AOP approach that provides dynamic weaving at the JVM level. As an extension of IBM’s Jikes Research Virtual Machine\(^5\) (RVM) (Alpern et al., 2005), Steamloom took advantage of the adaptivity of this RVM to add support for dynamic join point selection and runtime weaving (Bockisch et al., 2004). When introducing aspects, Steamloom does not require pre- or post-processing, since the VM is already responsible for integrating the advices and the execution.

Aspects are written in an aspect language similar to AspectJ, integrated with Steamloom, where aspects, pointcuts, and advices are defined as common Java classes. Steamloom provides a set of defined pointcut designator classes, and an advice consists of an instance of a class with the signature of a Java method and information regarding possible inputs. The method associated with the advice can retrieve information regarding the join point. An aspect is required to associate the advices to the pointcuts (Bockisch et al., 2004).

Steamloom follows the same deployment approach as CAESARJ\(^6\) (Aracic et al., 2006), which consists of triggering an aspect weaving using a deploy statement. An aspect can be deployed and undeployed with the provided methods in the Aspect class, with equivalent names (Bockisch et al., 2004, Haupt, 2005).

Steamloom provides two techniques for scoping an aspect: a thread-local scope, which allows the deployment of aspects for the scope of a certain thread; and an instance-local scope, applying an aspect only to a set of selected instances of a class. Therefore, the aspect deployment can be constrained to a thread or a class instance through an input argument (Bockisch et al., 2004).

The framework was built on top of RVM, to which it added a new API and a bytecode manipulation toolkit and extended the RVM intermediate-representation classes to support the weaving process. The RVM has a JIT compiler which can be executed without any optimization or modification, with the built-in optimizing compiler, or with an adaptive optimization system (AOS) for online profiling. Steamloom uses the AOS for RVM to support the dynamic join points and to take advantage of the optimizer compiler. The RVM records the loaded classes in an instance of a VM_class class. Steamloom uses the type information block (TIB), kept in the VM_class, to get references to the class methods, i.e., the VM_Method class instances. Static methods are accessed through the Jikes global table of contents (JTOC). These tables are used by RVM to produce a method lazy compilation when the method is invoked the first time (Bockisch et al., 2004).

When weaving an aspect, the Steamloom weaver modifies the target bytecodes with the bytecode manipulation toolkit and recompiles the bytecodes. Since some of the pointcuts may point to a method that could have been inlined by the optimizing compiler, the inlined versions should also be advised. Steamloom resolves the problem by maintaining a set of VM_Methods in

---

\(^5\) Jikes RVM website: http://www.jikesrvm.org/
\(^6\) CAESARJ website: http://www.caesarj.org/index.php
each method, containing the location of the method inlined version, forcing the weaver to apply the advices to those locations (Bockisch et al., 2004).

Since a TIB, in a class representation, refers to that class and hence to all the instances of the class, applying an aspect to only a set of instances of a class conflicts with the TIB approach. Steamloom solves the conflict by cloning the TIB and pointing the object instance to the cloned TIB version. Also when dealing with aspects that are applied to a selected thread and for supporting thread safety, Steamloom inserts small code snippets before every call to the advice functionality. This check defines if an aspect should be active in the current thread (Bockisch et al., 2004).

In summary, Steamloom provides a VM-supported dynamic AOP approach, which allows a more efficient runtime weaving, since the VM is directly prepared for receiving the aspect requests. The aspects are written similar to common Java programs, with a familiar approach to AspectJ. This approach not only allows the common redefinition of classes, but also to select a specific Thread the aspect can advise, or even only a collection of selected objects, instead of directly redefining their classes.

2.3.4 AspectWerkz

AspectWerkz (Bonér, 2004a) was developed to fulfill the need for a simple and powerful AOP approach that allows dynamic weaving and can be applied to real-world applications. AspectWerkz is focused on the design of a dynamic aspect model “in which the aspects, advices, and introductions are loosely coupled and are easy to add, remove or adapt aspects at runtime” (Bonér, 2004a). The dynamic weaving is done at class load time, a non-intrusive method for weaving a Java application (Bonér, 2004b).

The mechanisms of this approach consist of weaving, statically, the join points into the class and invoke the aspects, at runtime, by means of handlers. The concerning class is extended with a JoinPoint field. When the class is instantiated, the field is registered in the system and receives handlers to the advices. Once the join point is reached, the advices are invoked. The process has a small overhead applied to the class loading and instantiation and can (Bonér, 2004a):

- Add new aspects in the executing system;
- Associate new advices to a join point;
- Remove advice associations from a join point;
- Redefine advices;
- Add, remove and change introductions (described below).

An aspect in AspectWerkz is defined with a set of pointcuts, advices, and introductions. Aspects can be abstract, for instance with defined advices and abstract pointcuts, making them reusable by other aspects. For instance, in the previous example, one is able to define different aspects aiming at different pointcuts with the same advice, i.e., the same abstract aspect (Bonér, 2004a).

The pointcut definition follows the same methodology as in AspectJ, with join point selection and filtering with specific values and/or with wildcards. Join points are symbolized as instances of the class JoinPoint, which is available in the advice. The join point instance contains the information of the concerned point and advices are methods which execute over the join point
Background and Related Work

(Bonér, 2004a). The method takes as argument a JoinPoint instance. Like in AspectJ, AspectWerkz advices can execute before, after or around the join point. In the event of a before or after advice, no value is expected to be returned to the application and so these type of events return void. On the other hand, an around advice returns a Java object (java.lang.Object), since the application is expecting a value in the join point execution. A join point can have multiple advices concerning it. The execution of these advices takes a nested flow, in which the next advice is executed after the current advice terminates, or if the proceed method, included in the JoinPoint class, is invoked in the current advice. The method JoinPoint.proceed() allows one to carry on to the next advice targeting the join point, or the join point itself if the join point has no more advices (Bonér, 2004a).

An introduction, in AspectWerkz, enhances a Java class with a superclass, interface, method or field. For instance, when a logging process is to be added, an aspect can be defined with an introduction for a set of classes. This introduction adds a new interface and the implementations required for that interface. These interface methods can then be used, on AspectWerkz side, for logging, since this new interface is not recognized in the main application. The concept used in this framework for implementing the introduction is based on Mixins, where the inheritances, methods, and fields are defined in a mock class and later moved to the target class. Therefore, an introduction is defined as an inner class of the aspect class with the intended superclass, and perhaps new interfaces, to add to targeting classes (Bonér, 2004a).

Similar to DiSL, summarized in Section 2.7.2, a method tag, defined as a JavaDoc\textsuperscript{7} tag, specifies the pointcut to advise and where to weave. The advice can take place around, before (@before) or after (@after) the method. The after advice always take place, even if the method throws an exception. When the after advice should only apply if an exception is thrown or if successfully returns, the tags @after throws\textsuperscript{n} or @after returning\textsuperscript{e} can be used, respectively. The pointcut name, previous defined, follows the javadoc tag (Bonér, 2004a).

The introductions follow the same rule. Before a class declaration to be used for insertion, a tag @Introduce is added, followed by the pattern to the intended classes (Bonér, 2004a).

To define a scope in which an advice or introduction takes place, AspectWerkz offers the definition of a deployment model for the intended action. The deployment model defines the life-time of the action in four different periods (Bonér, 2004a):

- For each JVM: @perJVM;
- One instance per class: @perClass;
- One instance per object instance: @perInstance;
- One instance per thread: @perThread.

AspectWerkz can do dynamic and static weaving, performed at bytecode level using BCEL (Bonér, 2004a, Bonér, 2004b). The dynamic weaving approach is based on hooking after the bootstrap class loader, allowing bytecode transformations over classes when loaded in the class loader. Although the weaving process can be applied at load-time and runtime, pointcuts can only be defined statically. Aspects defined and applied during the execution can only target the pre-

\textsuperscript{7} JavaDoc website: http://docs.oracle.com/javase/1.5.0/docs/guide/javadoc/index.html
defined join points (Bonér, 2004a). In January 2005, AspectWerkz and AspectJ joined forces, adding the dynamic weaving environment of AspectWerkz to the maturity of AspectJ. The merge of these frameworks resulted in a single AOP platform with the capabilities of both frameworks (Bonér, 2004a).

### 2.3.5 LLDSAL

LLDSAL (Payer et al., 2012) is a Low-level Domain Aspect/Assembly Language for program instrumentation in a Dynamic Binary Translation (DBT) system. The language allows the specification of code snippets at compile-time that is dynamically used at runtime.

The DBT framework is a tool that allows for on-the-fly program modifications. This framework instruments the application with the required features and aspects during the compilation phase so it can adapt the application at runtime. It runs alongside the application and dynamically generates low-level code – the aspects – to alter, for instance, the program control flow, to switch between the original code and code devised from DBT, or even to apply optimizations (Payer et al., 2012).

The dynamically generated code must be fast and cannot rely on regular runtime environments. Hence, the DBT needed a fast and flexible way to generate low-level machine code on-the-fly. Payer et al. proposed LLDSAL to fulfill these requirements. LLDSAL uses a dynamic assembly language to express aspects that will generate low-level code to be woven in the target application at runtime. LLDSAL is used in a DBT framework to implement the dynamic code generator, translator and weaving process, with the ability to generate code for control flow manipulation, collect profiling information or generate dynamic policies (Payer et al., 2012).

In LLDSAL any instruction of the target ISA can be used. The assembly language used (AT&T assembler syntax) is extended with additional semantics, to allow access to high-level data structures and variables and functions and extensions to redirect control-flow to dynamic locations. Figure 2.8 depicts an example of using LLDSAL. LLDSAL is used inside a regular program, allowing it to use runtime parameters for the code generation. The `BEGIN_ASM(ptr)` and `END_ASM` statements comprise the dynamic block (lines 10 to 13), while the `ptr` is the variable in which the generated code will be stored. The generated code can then be used as a callable function, as shown in line 15 (Payer et al., 2012).

---

8 The merge of AspectJ and AspectWerkz was announced in: [http://aspectwerkz.codehaus.org/index-merge.html](http://aspectwerkz.codehaus.org/index-merge.html)
Background and Related Work

```c
void eip_gadget () {
    char * eip ;
    char *(* code )( void );
    //code must be mapped executable
    code = (char *(*)(void))(mmap (0, 4096, \n                    PROT_READ | PROT_WRITE | PROT_EXEC,\n                    MAP_PRIVATE | MAP_ANON , -1, 0));
    //code generation modifies ‘ptr’
    char * ptr = (char *) code ;
    BEGIN_ASM( ptr )
        movl (% esp ), % eax
        ret
    END_ASM
    //execute generated code
    eip = code ();
}
```

Figure 2.8. A simple example of LLDSAL being used inside a function. The LLDSAL code dynamically generates a function to determine the current instruction pointer (source: (Payer et al., 2012)).

Figure 2.9 depicts the compilation flow of LLDSAL. This language is implemented as a small extension of GNU C compiler toolchain. After the application is preprocessed by the GNU C preprocessor, the LLDSAL process the aspects defined as LLDSAL blocks and then executes them in the GNU assembler to generate the machine code. The machine code is then passed to the `objdump` to transfer the code back to the source file. This source can be compiled using the standard GNU C compiler.

![LLDSAL compilation flow diagram](source: (Payer et al., 2012)).
2.4 Runtime Optimization Frameworks

Dynamic software compilation efforts have already considered the runtime execution of optimization strategies. Dynamic feedback techniques, such as (Diniz and Rinard, 1997), (Voss and Eigemann, 2001) and (Rauchwerger and Amato, 2006), enable programs to automatically adapt to different execution environments. The following sections present some of the most representative runtime optimization frameworks.

2.4.1 ADAPT

The Automated Decoupled Adaptive Program Transformation (ADAPT) framework (Voss and Eigemann, 2001) considers the use of a domain-specific language to program an iterative compilation process. ADAPT (Voss, 2001) is a compiler-support framework which provides dynamic and adaptive optimization processes, making use of compiler optimizations and accessible optimization tools. The ADAPT compiler, having as input a target application, the optimizations required and the heuristics for applying the optimization generates application-specific runtime systems. The framework supports dynamic compilation, allowing one to explore different implementations through “runtime sampling” (Voss and Eigemann, 2001).

ADAPT uses a DSL to define the heuristics to apply optimizations. The ADAPT compiler takes these heuristics, the optimization techniques, and the target application, and generates a runtime system to apply the user-defined optimization techniques. Additionally, an optimizer is generated, which can run in parallel to the application’s execution, or in a remote machine, in the event of an application running in a single-processor system. ADAPT makes use of this optimizer to decouple the dynamic optimizations from the application execution (Voss and Eigemann, 2001).

The search for dynamic optimization approaches using ADAPT can be achieved iteratively. I.e., after generating a set of optimized versions, their performance can be measured and the compiler can generate new versions based on those results. Dynamically creating optimizations requires the use of the dynamic information of an application and the system in which the application is running. This information, based on the application input and the machine parameters on execution time, is used to choose the best code section for processing the data. The system must ensure that the optimizing sequence is useful for the current data (and system) and must be able to evaluate its choices (Voss and Eigemann, 2001).

In ADAPT, when a code section is selected for optimization, and the execution reaches this code section, there are two paths the application can take. The first one is the use of an already existing best known optimized version of the code, while the second is the use of an experimental version to be monitored. For choosing which path to take, ADAPT uses a flag to decide if the best-known version is to be executed, or if an experimental version is ready to be executed and profiled to feedback its execution data to the optimization process. When an experimental version is considered the best-known version, the optimizer swaps the old version with the new version. Both best known and experimental versions can be swapped during the application execution, with no need to halt. This approach only implies a small overhead when checking the flag (Voss and Eigemann, 2001).

ADAPT applies optimizations at compile-time and at runtime. The first process of the ADAPT compiler is the selection of the code sections of interest to be optimized. With this
Background and Related Work

selection, and addressing the user-defined heuristics, the compiler produces the runtime systems, i.e. modified versions of the application with the two paths, previously mentioned. Figure 2.10, extracted from (Voss and Eigemann, 2001), depicts the ADAPT framework. The compiler receives as input techniques with the heuristics and the definition of the accessible optimizing tools/compilers and the target source code application. A modified runtime system is generated, containing control paths in the code sections marked for optimization and a local optimizer, running in a separate thread, responsible for hot-spot detection, i.e., the most time-consuming code sections. The local optimizer communicates with the remote optimizer, which provides the former with the newly available code variants. ADAPT relies on previous behavior to generate new code. The remote optimizer has access to the source code candidate intervals for optimization, information regarding the target machine and the optimizing tools/compilers. Based on the heuristics and the previous results, the remote optimizer generates new code variants, which are compiled and stored in a shared library. The remote optimizer instructs to use the code section as the best-known version or as an experimental version. When the hotspot is reached, the new version is already available and ready to execute.

Figure 2.10. ADAPT framework during compile-time and runtime (Source: (Voss and Eigemann, 2001)).

The ADAPT language is a C-based language with special ADAPT statements and reserved words that define runtime data input or machine information, such as $l2$ size for referring to the L2 cache size of the machine (Voss and Eigemann, 2001).

ADAPT provides the development of adaptive optimizations using stand-alone optimization tools and compilers, available to the user. Through the user-defined heuristics and target application, the ADAPT compiler generates runtime systems which apply the heuristics dynamically. The tool avoids critical path decision overhead by decoupling the runtime optimizer from the application, running it in a free processor, or a remote system. The framework can be used in serial and parallel programs, being more suitable for the last.
2.4.2 rePLay

The rePLay framework aims to improve application performance following a practice of execution-guided optimizations. The idea combines instruction patterns with branch prediction (Patel and Lumetta, 1999). This framework takes advantage of runtime stability, i.e., instruction and data patterns that repeat during program execution. These patterns are used to generate frames, the base of the rePLay approach.

The framework uses optimized frames (single-entry, single-exit code regions) that replace the execution of normal, conditioned instruction sets. In a sense, a frame is a set of instructions, belonging to several basic blocks, which are completely executed, and hence no conditional expressions are used. Instead, ASSERT instructions replace conditional expressions. A missed ASSERT will produce a machine fail signal, and the application is returned to a recovery point. Some of these frames are used only in a single execution, while others may prevail for future executions (Patel and Lumetta, 1999).

Frames are studied and generated at runtime, using a Branch Promotion technique, where the rePLay framework analyses patterns that occur at runtime and, whenever a certain pattern is visited a specific number of times, the Branch promotion defines this pattern as a new frame. The Frame Constructor generates the frame with the given pattern, and an Optimization Engine improves this frame (Patel and Lumetta, 1999).

In order to use the generated frames, rePLay includes a sequencing mechanism, consisted of a conventional branch predictor and a frame sequencer. The selection is then between one of the original branches and the initiation of a frame (Patel and Lumetta, 1999).

2.4.3 SmartApps

SmartApps (Rauchwerger and Amato, 2006) uses an application-centric computing approach, creating what the authors call as “smart applications” that auto-optimize their execution, at runtime. This optimization may occur at the application and/or system level. SmartApps compiler adapts an application to embed runtime system services and a feedback loop to monitor performance and trigger runtime adaptations, thus creating a smart application. At runtime, giving the input and determining the system state, the SmartApps performs an instance-specific optimization. Parallel to the execution, the application keeps a continuous performance optimization and resource availability analysis to determine if a restructure should occur. The feedback loop monitors the performance and, based on the deviation of the expected performance, applies actions with the potential to improve the program efficiency (Rauchwerger and Amato, 2006).

This framework aims to perform “on-the-fly decisions” about optimizations and restructuring at different levels: algorithm adaptation, runtime optimization, operating system (OS) reconfigurable services and system configuration. The algorithm adaptation allows the application to choose between different algorithm implementations, one version that suits better for the current situation (e.g., choosing a sorting implementation). Runtime optimization can provide actions such as runtime parallelization and compiler optimizations, while reconfiguring OS services facilitates the changes in scheduling policies, page size, among others. Configuring the system allows the selection of computation resources (Rauchwerger and Amato, 2006).
Background and Related Work

A smart application consists of a runtime library embedded in the application and in the SmartApp compiler dynamically selecting which compiler optimizations are applied. This selection is based on three principles: maximize parallelism and minimize overhead computation, minimize wait-time of load imbalance, and minimize wait-time of memory latency. Additionally, SmartApps provides “non-intrusive architectural reconfiguration and operating system level tuning (…) to obtain fast, low overhead performance improvement” (Rauchwerger and Amato, 2006).

2.5 Algorithm Selection

The selection of the best algorithm is an approach to achieve the best performance for specific execution states. Having multiple algorithms with the same objective, each with advantages for specific circumstances allows a selective approach based on runtime information. There are several approaches using algorithm selection, such as (Wernsing and Stitt, 2010) and (Hao and Rauchwerger, 2006). These approaches use the best performing algorithm from a library of specialized algorithm versions. On the other hand, the approach of Hao Yu et al. (Hao and Rauchwerger, 2006) uses algorithm selection as an implicit methodology in their programming model.

2.6 Domain-Specific Languages for Adaptation

Unlike general-purpose languages, a DSL is a language focused in a specific domain, designed and specialized to express programs for that domain. DSLs provide important support of expressiveness to deal with program adaptation, such as (Santos, 2014) and (Cardoso, Carvalho, Teixeira, et al., 2012).

The development of embedded applications is usually subject to constraints and behavior of the system resources. An application can also profit from runtime information for constraining its execution according, for instance, to the device battery. The application adaptivity to these aspects is an important step for defining a reliable and efficient embedded system.

2.6.1 A DSL for Programming Runtime Adaptivity

André Santos (Santos, 2014) proposes a DSL specific for programming runtime adaptivity, e.g., for context-aware applications. It provides runtime system adaptivity, over embedded applications, by the specification of adaptation strategies. With these strategies, it is possible to define adaptation behavior based on program parameter tuning and algorithm selection. Independent of the target platform, the DSL provides a high-level abstraction for specifying the runtime adaptivity strategies, allowing rapid prototyping of adaptable systems.

A system adaptation process is specified as one or more strategies, where the adaptation strategies define rules that produce software reconfigurations. A strategy is composed of declarations, operations, rules, and auxiliary code sections. Declarations define static information to be used in the adaptive strategies, such as variables and parameters required. Operations specify the system behavior and which events should trigger the strategy. A rule
section defines the tunings that will reconfigure the system. Code sections provide the definition of external functions defined in the supported languages (Santos et al., 2011).

Figure 2.11 depicts a strategy for adapting an application named StereoNav. The strategy adapts an image resolution according to the current vehicle speed. As mentioned above, a strategy can define declarations, specified at the strategy beginning. The example defines three declarations: an array with image resolutions, two functionalities defined in the target application (lines 2 to 7). The declarations are followed by the operation, defined in lines 9 to 13, which specifies a loop invoking the stereoNavStep method, in a location e1. Finally, the defined rule, in lines 14 to 26, adapts the image resolution, according to the vehicle speed, with the values in the image resolutions array (Santos et al., 2011).

A code section can also be added to a strategy. This section can provide new functionality to the system without any changes. The new functionalities are defined in the target platform language (Santos et al., 2011).

```
strategy StereoNavImResAdapt {  
  int[][] imgRes = {{320, 240}, {640, 480}};
  import function stereoNavStep(
    int ransacIterations=5000,
    int imageWidthRes=640,
    int imageHeightRes=480);
  import function [int vehicleSpeed] = getVehicleSpeed();

  operations() {
    // location where the rule will be triggered
    r1.evaluation_point('e1');
    stereoNavStep(ransacIterations, imageWidthRes, imageHeightRes);
  }

  rules {
    // adapt resolution according to vehicle speed
    r1: every(stereoNavStep): // at every stereoNavStep repetition
      retrieve getVehicleSpeed().vehicleSpeed as speed {
        if(speed > 50) {
          stereoNavStep.imageWidthRes = imgRes[0][0];
          stereoNavStep.imageHeightRes = imgRes[0][1];
        } else {
          stereoNavStep.imageWidthRes = imgRes[1][0];
          stereoNavStep.imageHeightRes = imgRes[1][1];
        }
      }
  }
}
```

Figure 2.11. Adaptive strategy redefining an image size, according to the vehicle speed (Source: (Santos et al., 2011)).

A strategy can define multiple rules and use other strategies. This provides higher abstraction levels when defining a strategy using formerly defined strategies, selecting which strategies are deployed, deactivate strategies, overriding strategies, etc. The strategies defined in this DSL are translated to the targeting embedded system language (currently the system supports C and Java languages). A strategy can be executed in parallel to the target application, which captures application events to the defined operations (Santos et al., 2011).

---

StereoNav is an embedded-subsystem responsible for vehicle localization when the satellite system of the vehicle is not available.
2.6.2 DiSL

Marek et al. (2013) propose a DSL for instrumentation, named DiSL, in the context of dynamic program analysis. It is a language based on the ASM bytecode manipulator framework (Bruneton et al., 2002) linked with a toolkit for instrumentation named jBORAT (Java Bytecode Overall Rewriting and Analysis Toolkit). The language contains an “open join point model where any region of bytecodes can be a shadow” (Marek et al., 2012). This framework focus on the importance of instrumenting the target application, making use of both static and reflective context information retrieved from a join point shadow, and how to provide information to the target code in an efficient way.

DiSL has a higher abstraction level to define the advices, maintaining a low-level approach for expressing new markers, the tool representation of pointcuts. These pointcuts can be defined using pre-defined markers, including method body, exception handlers, and single bytecode instruction. When the pre-defined markers are not specific enough or lack a required set of bytecode instructions, the tool provides constructors to generate new user-defined markers. Since it is an embedded language, having Java as the host language, markers are symbolized as standard Java classes that implement a Marker interface. DiSL provides the usual static context information, regarding properties, for instance, of classes and methods, and additionally offers information concerning the applications basic blocks or the bytecode instructions. Furthermore, the tool offers static analysis which can be defined by the user and computed at weave-time. The dynamic context can be exposed using a reflective Application Programming Interface (API) that can access local variables and the operand stack (Marek et al., 2012).

The advices are defined as Java methods, containing the code snippets to inline before, or after, the selected markers. The snippets can be weaved in any Java class, including the standard libraries of Java since the weaving is accomplished over the bytecode instructions. A method body, containing the advising code, is used as a template to be instantiated and inlined in the target join point. The insertion order of this snippets, for the same join point, is defined with an attribute of the method annotation which takes an unsigned integer. The lower the value, the closer the snippet is to the join point. A marker, i.e. pointcut expression, is also specified as an attribute of the previous tag annotation. The weaver captures the marker and instantiates it as an instance of a Marker class (Marek et al., 2012).

Both static and dynamic context information can be retrieved in DiSL. The static context can retrieve information, gathered at weave-time, regarding a class, a method, a basic block or even a bytecode instruction. Dynamic context retrieves information about the local variables, the operand stack, or the current object (in Java: this) in which the instrumenting join point resides. (Marek et al., 2012).

DiSL includes two mechanisms to limit where the snippets are weaved. Guard is a weave-time condition that evaluates a restriction for each join point. Scope consists of a snippet annotation attribute that takes a signature pattern to specify a method, class or package. Therefore, a guard is usually used when a fine-grained selection is required, whereas a scope is for fast method filtering (Marek et al., 2012).

2.6.3 Elastic Computing

Elastic computing (Wernsing and Stitt, 2010) is a framework that tries to separate program functionality from the implementation details. The concept is based on using specialized, elastic,
functions that allow an optimization framework to try different possible implementations, including an implementation that use different algorithms for the same functionality (Wernsing and Stitt, 2010).

This framework uses a library of specialized elastic functions, a tool for implementation planning and a runtime environment system to combine with a given application code. The user only needs to specify the function, without specifying how the elastic function used is implemented. Instead, the framework is responsible to dynamically choose the best implementation. For instance, if one intends to use a sorting function, it only needs to call the elastic sort function. At runtime, the elastic function invokes the elastic computing system to analyze runtime parameters and select the most efficient implementation, according to the profiles statically determined by the planning implementation tool (Wernsing and Stitt, 2010).

2.6.4 Adaptive Algorithm Selection for Reduction Parallelization

Hao and Rauchwerger (2006) developed a framework for algorithm selection for reduction parallelization to solve issues with data-sensitive algorithms containing irregular memory management (e.g., irregular accesses and dynamic allocations). The framework reckons an automatic selection of the best-performing functionality, from a library of candidates. It is based on an offline generation of predictive models, according to a set of memory references characteristics, and the runtime selection based on the current loop characteristics.

2.6.5 PetaBricks

PetaBricks (Ansel et al., 2009) is an “implicit parallel language” for program development based on algorithmic choice. In this language, the programmer specifies multiple algorithms solving the same problem, without the designation of which version should be used for each situation. Instead, based on those algorithms and some experimental testing accomplished by the PetaBricks compiler, an optimized hybrid algorithm is generated and autotuned. The compiler is able to generate hybrid algorithms combining the defined algorithms. During execution, the application is able to select the best approach for the current situation.

2.7 Template-Based Code Generation

Code generation is another well-known practice (Herrington, 2003) that allows automatic development of specific code. This concept is used in several approaches, whether for automatic code generation for model-driven development (Selic, 2003), for the automatic design for embedded processors (Leupers, 2000) or even for algorithm specialization (Charles et al., 2014). Many of these approaches are based on the use of predefined code snippets, usually called templates. These templates allow the generation of code with specific functionality, dedicated to a range of input values. Some of the current state-of-art approaches use offline code generation for developing multiple code versions, such as (Veldhuizen, 1998), or even for hardware design, such as circuit generators (Chu et al., 1998, Mencer et al., 1998). Runtime code generation, such as (Charles et al., 2014) and (Khan et al., 2008), allows the generation of specialized functionalities with execution-related information.
Background and Related Work

2.7.1 Circuit Generators

The use of circuit generators (see, e.g., (Chu et al., 1998, Mencer et al., 1998)) for hardware design automation has similarities to the approach of template-based code generation. The circuit generators approach by Chu et al. (Chu et al., 1998) was developed as a Java library. Following an object-oriented approach, the design is based on the specification of classes, inherited with specific base-classes, which represent the circuit components and their related connections. A class, defining a specific subcomponent, extends the `GenComponent` class. This class contains slots that define the inputs and outputs relative to this component, and also slots for subcomponents that give structure to the design. The circuit is generated based on the structure designed with the defined objects. The approach includes a partial evaluator to generate specialized designs. By accepting parameters in the design, the generator restructures itself to form a specialized instance. The partial evaluation assigns inputs with the known values and the components, using the assigned connections, examine the inputs and rearrange their internal structures, calculate the outputs and even remove components no longer necessary.

PAM-Blox (Mencer et al., 1998) is also an object-oriented circuit generator, developed in C++ on top of peripheral component interconnect design environment named Pamette. In this approach, the design is abstracted into two layers: `PamBlox` and `PaModules`. `PamBlox` defines simple parameterizable templates of hardware as objects (e.g., counters and shifters). `PaModules` define complex, fixed circuits as objects. A `PaModule` is defined with multiple `PamBlox` optimized for specific data-width. The design of data-paths for FPGAs is implemented as an object-oriented hierarchy, ideally for the designers to take advantage of C++ object-oriented approach, using function overloading, virtual functions and templates for a more efficient design. With this, the designer has total control over the placement of each level of the design hierarchy. This approach encourages code-reuse for future designs and also code-sharing among the PAM-Blox designers community (Mencer et al., 1998).

2.7.2 deGoal

deGoal (Charles et al., 2014) is a tool designed to provide the capability of implementing functionality that is tunable at runtime, according to runtime information, such as execution context and target processor. Since at compile-time it is not possible to produce machine code on the basis of execution context knowledge, deGoal provides a tool that embeds dynamic code generators into applications, providing runtime data-depending optimizations for the target processing kernels (Couroussé and Charles, 2012, Charles et al., 2014). It is able to perform cross-JIT applications, where a compilette runs on one processor model and generates code for another processor, downloading the generated code into the application.

The tool works over the applications kernels, the most computationally exhaustive part of an application. The kernels are built-in fast, portable, and small, binary code generators called `compilettes`, with only a few kilobytes. A `compilette` generates ad hoc versions of a kernel code at runtime that are optimized to the current program/system situation (Couroussé and Charles, 2012).

`Compilettes` are defined in a deGoal language that combines standard C with a high-level ASM language. This language provides instruction parameterization, with values specified at runtime. deGoal compiler translates the written compilettes into pure C source files and the complete application may then be compiled by a standard C compiler. During the execution of
the application, the compilette(s) generates optimized binary code versions of the target kernel, ideally in a different processor than the one executing the application. deGoal is able to produce compact code, as the specialized part is the only required to be generated, and not all variants are generated at the same time (Couroussé and Charles, 2012, Charles et al., 2014).

### 2.7.3 HySpec

When applying compiler optimizations, it is possible for the program to suffer from code explosion, having well-optimized versions with the cost of large code sizes. Dynamic specialization can use runtime information, however, it requires runtime compilation tasks, which may degrade the performance of the application. Khan et al. (Khan et al., 2008) propose HySpec (Hybrid Specialization), an automated approach to deliver code specialization at runtime. HySpec intends to address the problems existent in compile-time optimizations and dynamic specialization (Khan et al., 2008).

The proposed concept overcomes the code size and runtime activities overhead with a hybrid specialization approach, by first generating optimized code, through specialization at compile-time, and then generating templates working with a set of input values, through runtime specialization. This runtime specialization is performed for a small number of instructions, in a generic binary template. The templates are generated at compile-time, with optimizations applied based on the definition of values, unknown at compile-time. During the application execution, these templates are adapted to new values (Khan et al., 2008).

Hence, this approach does not generate completely different codes. Depending on the value ranges, the codes have the same instructions and only differ in some used constants. The value ranges are required to obtain previously to the use of the approach, which can be obtained by profiling, user-input, or data analysis. A binary template is then generated for the input ranges, and when instantiated, during execution, with a specific set of parameter values, it is equivalent to the versioned code (Khan et al., 2008).

The flow of this hybrid specialization approach is shown in Figure 2.12. It starts with versions of the selected function for a set of parameters’ values. For each version, a template is developed when possible, generating a dynamic specializer together with specialized data for that template. The final hybrid code has the versioned templates, the developed dynamic specializer and the original code if a fallback is required (Khan et al., 2008).
HySpec takes an input configuration file describing the target functions, the required parameters, value intervals, and the compilation parameters. The intervals may be defined using application knowledge, or with HySpec compiler, instrumenting the target code to obtain the required values.

2.8 Frameworks for Runtime Adaptivity

The following approaches represent other runtime adaptation tools used in different concepts, objectives or systems.

2.8.1 ASM

ASM (Bruneton et al., 2002) is a Java class manipulator tool which allows dynamic changes using bytecode manipulation practices. The ASM methodology uses a visitor design pattern to represent and visit the bytecodes relative to an object. Other tools using the pattern, such as BCEL and SERP\(^\text{10}\), represent the bytecode instructions as object instances. This representation leads to a large set of classes to symbolize the bytecodes and instantiations required for each bytecode occurrence. The object representation requires a tool with a lot of classes, which adds a lot of memory space and execution time overhead for instantiating the bytecode image. BCEL contains around 270 classes for representing the bytecode instructions, and SERP consists of 80 classes for the same purpose. ASM follows the same visitor design pattern, however, it does not symbolize the bytecodes with objects. The representation is reduced to only 13 classes (Bruneton

\(^{10}\text{SERP website: http://serp.sourceforge.net/}\)
et al., 2002). This avoids the creation of objects with short-life since these objects are only used during the bytecode manipulation and then deleted after their short-time purpose.

The ASM approach can visit a serialized object graph, without complete deserialization, and modify the graph structure. In order to work with the complex steps of serialization and deserialization of a class bytecode, ASM provides automatic processes to manage the class constant pool and its structure. The framework uses labels to manage the instruction addresses when inserting new instructions between two existent instructions and also provides utensils for computing the maximum stack and the local variables of a method (Bruneton et al., 2002, Kuleshov, 2007).

The practice of using ASM involves the implementation of a set of visitor interfaces for classes, methods, fields or annotations, providing access to any bytecode instruction. Each visitor has a specific target part on a Java code. Using a visitor design pattern allows the creation of simple local transformations that do not require context. By defining more complex visitors, chaining different visitors for different purposes, one can develop more complex transformations (Bruneton et al., 2002). In order to completely read a class or method already in memory, ASM also provides a tree package to manipulate the class, or method, as a Document Object Model (DOM) tree representation. The representation contains all the available data of the visited node and allows the redefinition of the tree structure (Kuleshov, 2007).

ASM can be used for a vast set of modifications in a target application during class load-time. These modifications can be applied in the class structure, such as introducing interfaces and superclasses, add new fields and methods, replace the existing method or field, etc. The advice at the class level is the common approach for many tools, such as PROSE (Popovici, 2003.) and AspectWerkz (Bonér, 2004a). ASM provides more fine-grained advice, allowing changes to a field or even inside a method. It is possible to insert new code before any bytecode instruction, inline a method or replace field access or method call (Kuleshov, 2007).

2.8.2 Dynamic adaptivity in LLVM

There are some approaches targeting the low-level virtual machine (LLVM) compiler infrastructure (Lattner and Adve, 2004) for dynamic adaptation. Two examples of these approaches are the ones presented in (Engel and Freisleben, 2005) and (Lomüller and Charles, 2014), aiming for dynamic instrumentation and runtime program specialization, respectively.

TOSKANA-VM (Engel and Freisleben, 2005) is an AOP approach that provides dynamic weaving by means of LLVM. The TOSKANA-VM system, shown in Figure 2.13, is composed of an L4 microkernel connected to the system weaver and a set of LLVM instances, associated with LLVM program bytecodes. The microkernel is responsible for splitting an application functionality between the available system components, being also responsible for memory and task management and inter-process communication (IPC) primitives. The Weaver is responsible for triggering the join point selection of a target LLVM instance, instructing it to intercept an instruction execution. The LLVM offers control over the execution instructions, the bytecodes of LLVM, which can be used for aspect-weaving. Each LLVM instance can run an L4 Linux or an application compiled to the LLVM bytecodes (Engel and Freisleben, 2005).
Background and Related Work

TOSKANA-VM join points are defined by the type of instructions available in LLVM. Currently, the platform provides access to a method call/execution and variable assignments/access. The LLVM execution is halted while the advice is invoked and executed. The advice is compiled with a JIT compiler. Returning from an advice resumes the LLVM execution (Engel and Freisleben, 2005).

The LLVM extension, still in development by Lomüller et al. (Lomüller and Charles, 2014), has the purpose of generating low-overhead runtime program specializers. This retargetable runtime program specializer generator is independent of any high-level language and can target embedded platforms. It relies on an intermediate representation (IR), thus allowing complete independence to use the generator with any LLVM front-end. Another objective of this approach is the “cross-specialization”, which is the specialization of a routine for a different platform from the one that is performing the specialization.

The approach starts with a program including annotations (LLVM intrinsic, i.e., pseudo-function call), where the annotations identify variables and the code generation locations. The program proceeds through a set of dedicated passes to analyze static values and static, dynamic and semi-dynamic operations. The code generation is executed to analyze what should be dynamically generated, creating trackers that are in charge of gathering information. This process outputs templates for the code to be specialized at runtime, and a database with the information of the trackers. The code generator merges these outputs to produce the final code for the LLVM-IR.

2.9 Overview

The studied state-of-art approaches enclose a variety of different goals (e.g., instrumentation or optimization), methodologies, target languages and/or environments, in order to adapt a target application. Table 2-I, Table 2-II and Table 2-III show a comparative analysis between those approaches, regarding how the strategies are specified, the target system, the weaving type, the grain level selection and important observations that stand out. Table 2-I includes the LARA language in order to compare the approach proposed in this thesis with the starting point, i.e. the first LARA approach (Cardoso, Carvalho, Coutinho, et al., 2012).

Some of the state-of-art approaches target general-purpose systems and applications, with no concern over the system in which the application executes. These approaches can be applied
in general-purpose concerns in which the application performance is not an important aspect. When application performance is required, as in most real-world applications, the system state becomes an important feature for applying performance concerns over the application, taking into account the available resources and several parameters. Moreover, the runtime data, only achieved during the application execution, provide important feedback for the tool and subsequently can be used, for instance, for code optimization or specialization.

Current approaches concerning these requirements, such as (Santos et al., 2011) and (Voss and Eigemann, 2001), provide important background knowledge when triggering adaptation to both embedded and general-purpose systems. Furthermore, they provide different specification types of runtime adaptation, in which the former provides a manual specification and the latter offers automatic systems generation through heuristic specifications. Manual specifications (e.g., (Santos et al., 2011), (Charles et al., 2014) and (Wernsing and Stitt, 2010)) provide more autonomy to developers to decide which code sections are considered critical and which type of adaptivity should take place. However, this requires good knowledge over the application and most certainly what should be applied to improve its performance, according to the code section “input” value. Automatic adaptivity definition (see, e.g., (Patel and Lumetta, 2001, Rauchwerger and Amato, 2006, Khan et al., 2008)) provides an easier method, requiring only, for instance, user-defined heuristics (Voss and Eigemann, 2001) so the system can find the hotspots in the application. However, in these approaches, developers have reduced control over the automatically generated system.

The most common weaving method when targeting an application running in a virtual machine (VM), passes through bytecode instrumentation, both for load and runtime weaving. Most of the approaches with runtime weaving require a specific, customized, VM that accept the weaving process (Bockisch et al., 2004, Engel and Freisleben, 2005, Würthiger et al., 2010). The most common action provided is the code/bytecode insertion, including, in some cases, class member modifications. The insert action is generally used for monitoring and debugging purposes, and can also be used to add new functionalities to the application. Code instrumentation is also used for information retrieval, used to provide runtime feedback for another tool.

It is important to have several types of actions for an approach to deliver runtime adaptivity, such as code specialization, optimization or mapping specific code snippets to the available CPUs or other devices. ADAPT is an approach that allows one to use tools available in the system that provides optimization mechanisms. LARA also provides flexibility to allow new types of actions, in an AOP approach.
Table 2-I. Comparison of the state-of-art approaches regarding DSLs for adaptation and Dynamic AOP.

<table>
<thead>
<tr>
<th>Aspect/Strategy Definition</th>
<th>Santos’ DSL (Santos, 2014)</th>
<th>DISL (Marek et al., 2012)</th>
<th>HotWave (Villazón et al., 2009b)</th>
<th>HotWave2 (Würthinger et al., 2010)</th>
<th>PROSE (Popovici, 2003)</th>
<th>Steamloom (Nicoara et al., 2008b)</th>
<th>AspectWerkz (Bonér, 2004a)</th>
<th>LARA (Cardoso, Carvalho, Coutinho, et al., 2012)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target System</td>
<td>General-Purpose</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Embedded</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Weaving</td>
<td>Time</td>
<td>Compile</td>
<td>Load</td>
<td>Runtime</td>
<td>Runtime</td>
<td>Runtime</td>
<td>Load</td>
<td>Compile</td>
</tr>
<tr>
<td></td>
<td>Source Code</td>
<td>Bytecode</td>
<td>Bytecode</td>
<td>Bytecode</td>
<td>Bytecode</td>
<td>Through RVM’s adaptive</td>
<td>Source Code</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bytecode</td>
<td>Source Code</td>
<td>Bytecode</td>
<td>Bytecode</td>
<td>Bytecode</td>
<td>optimization system</td>
<td>Bytecode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hotswapping</td>
<td>Through RVM’s adaptive</td>
<td>optimization system</td>
<td></td>
<td>Hotswapping</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Type</td>
<td>Source Code</td>
<td>Bytecode</td>
<td>Bytecode</td>
<td>Bytecode</td>
<td>Hotswapping</td>
<td>Source Code</td>
<td></td>
</tr>
<tr>
<td></td>
<td>System behavior</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Local Variable</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Method Call</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Method Execution</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Join Points</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Loop</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Conditions</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Class</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Field</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Method Definition</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Shadow Retrieve</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Static</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Dynamic</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Actions</td>
<td>Runtime adapt.</td>
<td>Code snippet</td>
<td>Code insertion</td>
<td>Code insertion</td>
<td>Code insertion</td>
<td>Code insertion</td>
<td>Insert, optimize, HW mapping…</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>inlining</td>
<td>and class redef.</td>
<td>and class redef.</td>
<td>and class redef.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Observation</td>
<td>Requires Own VM</td>
<td>Runs only in RVM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-II. Comparison of the state-of-art Runtime Optimization Frameworks and approaches for Algorithm Selection.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Target System</td>
<td>DSL</td>
<td>-</td>
<td>-</td>
<td>Library</td>
<td>Library</td>
<td>DSL</td>
</tr>
<tr>
<td>General-Purpose</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Embedded</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Weaving</td>
<td>Time</td>
<td>Compile &amp; Runtime</td>
<td>Compile &amp; Runtime</td>
<td>Compile &amp; Runtime</td>
<td>Compile &amp; Runtime</td>
<td>Compile</td>
</tr>
<tr>
<td>Time</td>
<td>Application-specific runtime systems</td>
<td>Frames Creation</td>
<td>Embedded Library</td>
<td>Elastic functions</td>
<td>Choice between candidates</td>
<td>Generation of Hybrid algorithms</td>
</tr>
<tr>
<td>System behavior</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Local Variable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Method Call</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Method Execution</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Field</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Method Definition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shadow Retrieve Static</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dynamic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compile-time weaving to define critical code sections</td>
<td>Instruction pattern + branch prediction</td>
<td>Runtime decision of best version</td>
<td>Offline Generation of predictive models</td>
<td>Use of experimental testing for hybrid generation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

45
Table 2-III. Comparison of the state-of-art: template-based code generation and tools for runtime adaptivity.

<table>
<thead>
<tr>
<th>Aspect/Strategy Definition</th>
<th>deGoal (Charles et al., 2014)</th>
<th>HySpec (Khan et al., 2008)</th>
<th>ASM (Kuleshov, 2007)</th>
<th>TOSKANA-VM (Engel and Freisleben, 2005)</th>
<th>LLVM Runtime spec. (Lomüller and Charles, 2014)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Target System</strong></td>
<td>C extended with high-level assembly</td>
<td>Config. File</td>
<td>JAVA Library</td>
<td>C procedure</td>
<td>Annotations</td>
</tr>
<tr>
<td>General-Purpose Embedded</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Weaving</strong></td>
<td>Compile &amp; Runtime</td>
<td>Compile &amp; Runtime</td>
<td>Compile</td>
<td>Compile &amp; Runtime</td>
<td>Compile &amp; Runtime</td>
</tr>
<tr>
<td>Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>Use of Compilettes</td>
<td>Templates</td>
<td>Source Code</td>
<td>LLVM instructions</td>
<td>LLVM-IR</td>
</tr>
<tr>
<td>System behavior</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local Variable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Method Call</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Method Execution</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Conditions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Field</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Shadow Retrieve</strong></td>
<td>Static</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dynamic</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Observation</strong></td>
<td>Use of embedded generators</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2-IV presents the most relevant characteristics the approach presented in this thesis intends to achieve, comparing some of the most relevant studied approaches with the intended approach, defined in the last row. The code specialization shows the use of code that is, for instance, specialized to a specific range of values, while algorithm selection is the possibility of providing algorithmic choice, between a list of candidates, at runtime. An approach that is context-aware is able to provide runtime information about the program, system and possible resources available. An execution-guided optimization approach allows program optimization based on profiling a continuous program execution.

The approach proposed in this thesis distinguishes from the studied approach by following the specified characteristics, with the efforts to define a DSL that follows an AOP approach able to express runtime adaptivity strategies and the dynamic application of template-based code generators. The approach is focused on the use of a DSL to express the application of runtime code optimizations, using, for instance, code specialization based on contextual information. The approach, regarding runtime specialization/optimizations, is orthogonal to the work on JITs.
Background and Related Work

(Aycock, 2003) as it provides a specialization layer, focused on adaptive strategies based on pre-optimized/specialized Bytecodes. The adaptivity is envisioned as strategies expressed in LARA that at runtime decide among specialized/optimized (predefined /generated) versions. Furthermore, the proposed approach includes an event-like method where one may define when the adaptation occurs, allowing developers to decide where, what and when should the strategy be applied.

Table 2-IV. Comparison of the most significant approaches with the main characteristics intended in this work.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Santos’ DSL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>(Santos, 2014)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LLDSL</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>(Payer et al., 2012)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADAPT</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>(Voss and Eigemann, 2001)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rePlay</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>(Patel and Lumetta, 2001)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>deGoal</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>(Charles et al., 2014)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmartApps</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>(Rauchwerger and Amato, 2006)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HySpec</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>(Khan et al., 2008)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PetaBricks</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>(Ansel et al., 2009)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LARA for Runtime Adaptivity</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

* Event handling considers the use of event-like approaches that allow strategy execution based on system/program events, such as a low-battery event or a timed adaptation.

2.10 Summary

This chapter briefly described the most relevant background and related work showed in Section 2.9 an overview and an analysis over the studied approaches.

The chapter briefly described LARA (Cardoso, Diniz, et al., 2013), a DSL centered on an aspect-oriented programming (AOP) approach (Kiczales et al., 1997), being the current approach mainly focused on offline changes (static weaving) to the application (Cardoso, Carvalho, Coutinho, et al., 2012, Cardoso, Teixeira, et al., 2012). LARA is an AOP language designed for specifying non-functional requirements (Cardoso, Coutinho, et al., 2012, Coutinho, Carvalho, Durand, Cardoso, et al., 2012), and a weaver that receives as input the intermediate representation of the aspects (Aspect-IR (Coutinho, Carvalho, Durand and Cardoso, 2012)). LARA allows developers to specify the non-functional requirements detached from the source code and in what
order these concerns should influence the code. This allows one to try different application designs (Coutinho, Carvalho, Durand, Cardoso, et al., 2012).

The work developed in the context of LARA provided the fundamentals and the motivation for the development of a new approach for runtime adaptivity which is based on LARA extensions.

The DSLs presented allow application adaptivity with different techniques and purposes. The DSL proposed by Santos et al. (2011) provides mechanisms to enable runtime adaptivity over an application. DiSL offers runtime instrumentation over Java applications for dynamic data analysis (Marek et al., 2012).

The AOP community has several approaches targeting dynamic weaving. For instance, HotWave (Villazón et al., 2009b), and HotWave2 (Villazón et al., 2009b) provide runtime adaptation allowing to weave previously loaded classes (Villazón et al., 2009b), where the second version has higher access and flexibility over any Java application, with the downside of only working in its VM (Würthinger et al., 2010). PROSE is capable of weaving an application in runtime without stopping the execution, being more suitable for applications for rapid prototyping (Popovici et al., 2002). Steamloom is a VM-supported dynamic AOP approach, allowing efficient runtime weaving, with aspects written in common Java programs (Haupt, 2005). Finally, AspectWerkz (Bonér, 2004a) can do dynamic weaving, performed at bytecode level using BCEL by hooking after the bootstrap class loader, with a static pointcut definition (Bonér, 2004a).

Moreover, the most relevant tools and frameworks allowing runtime weaving and adaptation were highlighted. ADAPT provides the creation of adaptive optimizations, using stand-alone optimization tools and compilers, with user-defined heuristics, generating a set of runtime systems which apply the heuristics dynamically (Voss and Eigemann, 2001). It is also depicted ASM, a framework allowing the visit of any bytecode instruction with a simple and high-level approach, allowing at the same time the insertion of low-level (bytecode) instructions (Bruneton et al., 2002). For the LLVM infrastructure, TOSKANA-VM (Engel and Freisleben, 2005) provides dynamic weaving for instrumentation, aiming the LLVM bytecodes (Engel and Freisleben, 2005), and Lomüller and Charles (2014) proposes a low-overhead runtime program specialist, independent from the target high-level language.

Some approaches, such as Wernsing and Stitt (2010), Hao and Rauchwerger (2006) and Ansel et al. (2009), provide specialized functions that allow an optimization framework to use different possible implementations, and the best implementation is dynamically chosen. The rePLay approach (Patel and Lumetta, 1999) practices execution-guided optimization, creating optimized frames that replace the execution of normal instruction sets. deGoal (Charles et al., 2014) implements functionality tunable at runtime, based on runtime information, by means of embedded dynamic code generators. SmartApps (Rauchwerger and Amato, 2006) builds “smart applications” that dynamically auto-optimize its execution, at the application and/or system level. It was also presented HySpec (Khan et al., 2008), an automated approach to deliver runtime code specialization, using binary templates.
3

Support for Runtime Adaptivity at Software Level

Contents

3.1 Requirements and Analysis................................................................. 50
3.2 A DSL + API Approach for Runtime Adaptivity....................................... 55
3.3 LARA Framework for Adaptivity and Targeting Object-Oriented languages........ 58
   3.3.1 Weaving Framework................................................................. 59
   3.3.2 Weaver Generator ....................................................................... 60
   3.3.3 Aspect Composition for Reusability and Multiple Target Languages........... 62
3.4 Execution Points .................................................................................... 65
3.5 Design of Strategies .............................................................................. 68
   3.5.1 Compile-Time Code Restructuring..................................................... 68
   3.5.2 Adaptation Description .................................................................... 70
3.6 Sequential and Concurrent Adaptation..................................................... 71
3.7 Goals and Metrics for Runtime Evaluation.............................................. 74
3.8 Summary ............................................................................................... 75
Support for Runtime Adaptivity at Software Level

This chapter presents and discusses the approach proposed in this thesis for specifying runtime adaptivity strategies, based on a DSL, a weaver engine and an API.

3.1 Requirements and Analysis

The specification of runtime adaptivity is a complex task that requires the code to be developed in a specific fashion. Although an application, at its early stage of development, might be designed taking into account the possibility of runtime changes, tangling the runtime adaptivity within the main application logic produces complex and intertwined code, which can be solved adopting a separation of concerns (Hürsch and Lopes, 1995). A mature application defined without runtime adaptivity in mind and later required to be aided by runtime adaptivity requires a process even more complex. Maintainability in this type of applications is an important issue and manual changes might be too complex and/or error-prone. Thus, a more automated approach is well seen in this case. Such an approach would need to allow code injection in order to add new functionality and to apply code transformations in order to restructure the code to expose runtime information and allow runtime adaptivity.

Consider an application in which a developer intends to extend the use of a sorting algorithm to enable a runtime selection of the fastest algorithm. There are many factors to consider in the but he/she also of algorithms (Rice, 1976), such as the availability of different algorithms with the same functionality, the type, the range, the number of data elements, and the properties of the system specifications. These factors make the selection of the best performing algorithm per possible scenario difficult. As many algorithms might be specialized to the working dataset, and the specialized version might be pre-compiled or generated at runtime, the selection becomes even more challenging.

A runtime adaptivity strategy requires reasoning about how and the means to: transform the code to prepare program regions for adaptivity and to expose relevant program/system information, select the algorithms to sample, control the available algorithms, allow sampling configurations and measurement APIs, and inject code to select, sample and measure the different algorithms. Figure 3.1 shows a strategy for the sorting example, divided into three phases. First, it is necessary to expose and transform the code to allow the adaptation and provide imperative information (lines 1 and 2). Then, add the code (possibly using an API) that provides the set of algorithms, generate new versions and add monitors and the adaptation manager (lines 3 to 5). Finally, inject the code responsible for the adaptation (line 6).

<table>
<thead>
<tr>
<th>Strategy SelectSortingAlgorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Acquire the collection size;</td>
</tr>
<tr>
<td>2. Change calls to sort with dynamic invocations:</td>
</tr>
<tr>
<td>3. List available algorithms and code generators;</td>
</tr>
<tr>
<td>4. Add execution time monitor;</td>
</tr>
<tr>
<td>5. Add code to manage algorithms and measurements;</td>
</tr>
<tr>
<td>6. Inject code to:</td>
</tr>
<tr>
<td>a) select an algorithm to sample</td>
</tr>
<tr>
<td>b) select the best algorithm if sampling is done</td>
</tr>
<tr>
<td>c) measure the algorithm execution</td>
</tr>
<tr>
<td>d) report measurements to the manager</td>
</tr>
</tbody>
</table>

Figure 3.1. A recipe for an adaptive algorithm selection strategy considering execution time as the metric to minimize.
In order to drive the research and engineering of the proposed approach, we collected the following user stories (identified herein by US#). These user stories have been collected from user information discussed in a number of projects involving both academics and industry (such as the REFLECT (Cardoso, Diniz, et al., 2011) and ANTAREX (Silvano et al., 2016) EU projects).

(US1) "As a developer, I want to specify application points (locations) and knobs so that I can apply runtime adaptivity rules on those points and to those knobs"

At the core of a runtime, adaptivity system is the specification of the runtime strategies and their impact during the application execution. These strategies should have access to information only attainable at runtime, which can be used, e.g., for code specialization (Voss and Eigemann, 2001, Charles et al., 2014) and autotuning (Ansel et al., 2014, Gadioli et al., 2015). While a static approach only considers the use of compile-time and profiling information, a dynamic approach has to deal with runtime information, both application- and system-related. A runtime strategy needs to deal with these possible changes, usually not considered at compile-time.

While deciding where runtime adaptivity should be applied, one needs a way of accessing and manipulating the target code. The first approach is to do this manipulation by hand. The code is restructured over the source code text and fully managed by the developer. This approach has many drawbacks, such as the code maintainability becomes difficult and code manipulation is very error-prone. A more automated way of restructuring the code is more feasible in this sense, more specifically a tool allowing code manipulation. It is more practicable when this tool is able to deal with a syntax tree representation of the source code. From syntax trees, one may access fine-grained points of interest in the code and obtain relevant data from the source code with more efficiency.

These tools are usually controlled by input arguments, configuration files or by scripting or programming languages. Being able to programmatically observe and/or adapt specific points in the source code improves maneuverability for the specification of runtime adaptivity strategies. Code manipulation is an important component of this type of tools. It is important to have ways to restructure the code and inject new code. Combining runtime strategies with compile-time code transformations would allow developers to build more complex strategies. For instance, one may design a strategy that first monitors the execution and after a period of time activates adaptation by generating a new version to execute.

(US2) "As a developer, I want to specify adaptivity rules and/or automatic search schemes so that I can experiment with and deliver runtime adaptivity applications"

The specification of runtime adaptivity may not be trivial. Not only the developer needs to specify where to adapt the program and how to adapt, but he/she also has to identify the rules to apply considering the runtime environment. These rules can be as simple as comparing runtime data with pre-defined conditions to make adaptivity decisions or as complex as having search schemes that provide different configurations to explore and/or use machine learning techniques to predict configurations.
Support for Runtime Adaptivity at Software Level

Configurations can be of several types. One may have a configuration that deals with a single operation knob or has a configuration that sets up multiple target knobs. A knob can be a system parameter, a software parameter, an optimization parameter or even a target algorithm. Usually, a knob can be configured with a wide range of values/algorithms and search schemes allow one to try different configurations from within the knob range or a set of possible knob values/algorithms.

With search schemes, it is possible to explore different types of configurations. The more basic search scheme is the exhaustive exploration of all values the knob can be assigned. This scheme may be too expensive considering that the range of values might be too wide. If one considers search schemes dealing with multiple knobs, the number of possible configurations might be simply too large to explore. Hence, it is imperative to have different types of search schemes that allow one to reduce the exploration space. Some examples of these schemes are: jump/ignore some values in the range, explore surrounding values starting from a specific configuration, or use a number of random configurations. This exploration space reduction might come with the drawback of not achieving the best configuration, but it may allow finding a configuration that allows achieving an acceptable approximation to the objectives of the adaptation.

In this case, it is necessary a mechanism that allows one to not only apply the adaptivity schemes but also to control the adaptivity behavior in terms of runtime decisions. Adaptivity might be specified based on pre-knowledge or by a runtime learning process that uses a search scheme to explore different configurations until a satisfying one is found. This specification could be completely defined by the developer, giving complete freedom to him/her, or by using a semi-automatic approach with some freedom degrees for the developer, or even by an autotuning approach configured by the developer but with runtime decisions left for the autotuner.

(US3) “As a developer, I want to have a separation of concerns approach so that I do not need to manually maintain multiple versions of the application code”

Handcraft changes require the adaptation code to be inserted directly in the code. This reduces maintainability and hardens the reuse of the strategies for different versions of the application. Specifying multiple strategies for the same application allows the developer to distribute different versions of the application. Furthermore, to have different strategies for the same target requires one to maintain multiple versions of the code. After a strategy is weaved in the source code it becomes very difficult to rollback. This would require some kind of code versioning to successfully stash the changes applied because of the strategy. An example of this type of code versioning is the use of #ifdef and #pragma clauses in C language. These clauses are spread throughout the source code to mitigate compiler transformations and to define specific code based on the target system and user-defined information. To manually explore different compiler properties is still difficult as the developer still has to change pragma clauses information manually or create multiple #ifdef statements with different properties, further convoluting the source code (Alves et al., 2006).

Defining strategies apart from the source code is more convenient. It allows trying different strategies over the same code and the specification of strategies in an incremental fashion without
Support for Runtime Adaptivity at Software Level

affecting the original execution of the application. This approach can be used for the initial state of development of an application and even on final distributions.

A way to achieve this is with a DSL able to query the source code, to drive code transformations and to specify injection of new functionality within the application. The DSL shall allow programming strategies for compile-time code insertions and transformations in order to deal with runtime adaptivity and runtime code generation.

(US4) “As a developer, I want to express reusable and modular runtime adaptivity strategies so that I can possibly apply similar strategies to different applications or to multiple versions of the same application”

Runtime adaptivity strategies become even more useful when they are developed taking into account that they might be used in different development stages of the application or even for different target applications. One may have to consider changes between the different versions of the same application and redesign the strategy to deal with those differences, including members renaming, members moved to another compound unit and other transformations. Instead of maintaining multiple versions of the strategy, the developer should design a strategy reusable enough for the different application versions. If an adaptation strategy is abstract enough, it even can even be applied over different applications.

Reusable strategies should be able to abstract the target points of interest of the code and the adaptation code itself, i.e., transformations, the adaptivity type and the search schemes to use. Abstracting the target source code can be as easy as using wild cards instead of specific names. More advanced strategies would allow one to target more generic versions of the target code, e.g. Types instead of Classes and Interfaces or executables instead of methods, constructors and anonymous classes, or accept the points of interest as the input of the strategy. A reusable adaptation code would target these abstract code elements and might accept abstract parcels of code to be completed for specific targets.

(US5) “As a developer, I want to express runtime adaptivity regarding application parameters, algorithm choice, and code specialization so that I can approach runtime adaptivity in different contexts and application domains”

It is of interest of an application developer to address runtime adaptivity considering nowadays heterogeneous embedded systems. There are different types of adaptation one may practice over an application, depending on the type of access one has in the application and the target system, including the runtime execution environment and the target hardware. One way to adapt an application is to tune specific parameters, knobs, to achieve the runtime adaptivity goal. These knobs would allow one to control software, system or hardware parameters. If transformation/generation tools are available, a knob can also be used to control them.

Knowing that the same functionality may be defined by different algorithms, adaptivity can also be around the algorithms to be used. Having multiple algorithms for the same ending and knowing that they might perform better in specific circumstances, one may define strategies that
Support for Runtime Adaptivity at Software Level

select at runtime the best algorithm for those circumstances. It would also be necessary a way of using the exploration schemes if the performance of the algorithms is unknown.

Besides having multiple algorithms, a specific algorithm can also be optimized to execute more efficiently for a set of input values. This process is called code specialization and can be achieved by applying compiler optimizations to specialize the algorithm to specific input values/ranges (configurations). A runtime adaptivity approach should allow the generation of these specialized versions at runtime.

A runtime adaptivity has a goal, whether for execution time performance of an application, for optimizing energy consumption, for accuracy or a combination of multiple goals. These goals are dependent on the developer intentions and should be defined by him/her.

Hence, the approach shall consider these three different levels of adaptation and allow user-defined goals, e.g., regarding execution time and energy consumption.

(US6) “As a developer, I want to have an approach with minimal extensions to the programming language of the application so that it will be easier to learn and faster to use”

The strategies should be defined, preferentially, in the same programming language as the target application or in derivations of it. Demanding that developers must use a completely different approach than the target language requires a larger learning curve before starting programming and applying adaptivity strategies. This might delay the development of the application and reduces developer productivity. The approach should use a language close to the target language or with the same/similar functionalities so it is easier to learn how to develop strategies. In the case of a DSL not completely similar to the target language, then this language must provide enough similar functionality to ease the learning curve.

If the code to inject is completely written in the same dialect of the DSL, and assuming that this DSL is not similar to the target language, then the developer must learn how to completely write the expected code in that DSL, i.e., it must first translate the code to be injected to the DSL dialect and expect that same result when re-translating the code. This DSL to target language translation might be limited to a small set of constructs which reduces the type of code to inject.

Usually, it is more advantageous if the code of the strategy to be injected in the application is completely written in the target language, instead of using the DSL-related code. This means allowing the injection of native code and/or the use of API features that builds the necessary native code to be injected, instead of translating the DSL code to native code or to runtime execute the DSL code.

(US7) “As a developer, I want to use existing runtime adaptivity and exploration APIs and tools, as well as third party features that aid the development of adaptivity strategies”

Fixing the use of a single tool to deliver runtime adaptivity may not suffice, considering that there might be missing features that other tools could provide and complement the runtime adaptivity framework. If a new version of the tool is deployed or a more powerful tool is found,
then there must be a pragmatic way of replacing or extend the current working environment. In this sense, it is important to provide flexibility on the API to be used.

Based on these seven user-stories, the proposed approach to provide runtime adaptivity is centered on the use of a DSL with a framework responsible to apply changes in the application, including offline compiler transformations and optimizations, and an API providing runtime adaptivity strategies. The following section describes the proposed approach and how it answers to the listed user stories.

### 3.2 A DSL + API Approach for Runtime Adaptivity

The user-stories were the base on the decisions made for the proposed approach. \( U1 \) says that it is necessary to be able to select and access specific locations in the source code and to manipulate those locations to add the runtime adaptivity strategies. Two of the most fitting approaches for selecting and manipulating the source code is the use of code annotations or to use aspect-oriented programming (AOP) based approach. As \( U3 \) requires a separation of concerns, i.e. maintain secondary concerns apart from the main source code, then an AOP approach is more fitting. With AOP the target code is kept clean and the selection of pointcuts and code transformations are specified in the AOP language. This approach also allows one to easily parametrize aspects in a way that different code versions can be obtained from the same base code. This separation, and making use of abstract models and wildcards usually available in AOP approach, allows one to answer to \( U4 \), which intends to apply the same aspect over different target applications or even in different versions of the same application.

\( U6 \) reduces the problem to approaches that use a DSL that is not very different from the target language. As the previous decisions reduce the problem to AOP concepts, a first selection for the proposed approach would be the use of an AOP language that targets a specific programming language. These AOP languages add minimal extensions to the target language to allow the selection of points in the code, to advise those points with extra executable code or to introduce new structural elements in the application. Some examples of these AOP languages are AspectJ (Kiselev, 2002) for java, AspectC++ (Spinczyk et al., 2002) for C/C++ and AspectMATLAB (Aslam et al., 2010) for MATLAB. However, these languages are limited to a specific set of join points and do not natively provide access to fine-grained join points. For instance, AspectJ can access method execution and a field reference but cannot access the use of local variables or an if statement. Furthermore, these languages focus only on inserting new elements and adding new code around the target join points (e.g. add a new field). The specification of a runtime adaptivity may require the access to specific fine-grained locations in the target language and more advanced code transformation beyond code injection (e.g. loop unrolling).

Taking this into account, the required DSL should have similar constructions to the target language, should be able to select precise code locations in the target code and should advise those code locations with code injection or other types of code transformation. In the case that the DSL does not have access to a specific point in the code or a new code transformation must be added, then the DSL must be easy to extend with new AOP-related features.
Support for Runtime Adaptness at Software Level

The proposed researched and developed approach uses the LARA language for the specification of runtime adaptivity strategies. Although LARA has been used and shown for different contexts and goals (see, e.g., (Cardoso, Nane, et al., 2011, Cardoso, Diniz, et al., 2013)), as previously mentioned in Section 2.2.1, its use for runtime adaptivity has not been fully addressed. The versatility of LARA, for accessing fine-grained locations in the code and for code injection and transformations, provides the necessary support for the development of strategies able to perform specific runtime decisions to achieve different goals by means of code and parameter adaptation. To smooth the use of the proposed approach, it was decided to allow developers to express runtime adaptivity rules and search schemes for autotuning in the target programming language embedded in LARA insert constructs, instead of an approach where these concerns were specified fully using LARA native code. This would require more knowledge of LARA at the very beginning, a translation of LARA to the native code or a dynamic weaving approach which would impose non-negligible overhead.

The state-of-art work done for runtime adaptivity and autotuning provides important features which could be used within the LARA approach. Hence, it was a concern not only to propose an approach providing specific runtime adaptivity mechanism but also to allow strategy developers to use other existing adaptation techniques within LARA strategies. LARA is used in this context not only as the mechanism to control and guide the source to source compiler but also as the middle-end to add state-of-art adaptivity features within an application, by means of the aspect composition and modularity techniques (Pinto et al., 2018).

Although the proposed approach may be extended to the different target languages that LARA may target, this work focused on the runtime adaptivity in the context of the Java programming language. Java is one of the most used programming languages for developing applications (see, e.g., (TIOBE, 2019)), including the mobile domain. By providing a runtime adaptivity approach for Java one is also providing a portable adaptivity scheme. The runtime adaptation is at the Java Virtual Machine (JVM) level, providing a layer between the virtual instructions and the processor instructions. The adaptation in a virtual machine including a just-in-time compiler (JIT) provides benefits at the runtime compilation level and in the fact that a JIT approach enables the re-optimization of the program to the new specifications (Aycoc, 2003).

Figure 3.2 shows the flow of the proposed LARA-based runtime adaptivity concept for the Java programming language. At compile (or load) time the weaving engine, named Kadabra, processes the application to be adapted and the LARA strategies. The application is weaved where adaptation is intended, and where the required data can be retrieved, i.e., execution site where the necessary runtime information is collected. The engine generates a weaved version of the application that includes the Runtime Adaptation API requests specified in the strategy. Then, the application can be executed in a standard way in the JVM. During the program execution, when an adaptation request is triggered (e.g., specialization), the Runtime Adaptivity API, possibly in parallel to the system execution, generates a new version, according to the given input from the application and the provided code templates.
Table 3-I shows how the LARA DSL, the Kadabra weaver and the runtime adaptivity API addresses to the user stores specified in the previous section. The LARA language has the meanings of specifying the target join points and the actions to apply and the Kadabra Weaver is the one responsible to select the pointcuts and advise them. LARA aspects allow the specification of modular strategies that, if properly defined, can be reused for different application versions and other applications. To define reusable aspects in LARA one can use different features, from wildcards in the specification of target join points to the abstraction of the actual code to execute over those join points. The programming paradigm is very close to Java, since LARA uses JavaScript as its basis, and allows injection of native Java code. The main difference is, in fact, the select-apply paradigm that works very similar to query languages (see, e.g., SQL (Date et al., 2014) and XML-QL (Deutsch et al., 1999)).

A Java Runtime Adaptivity API was developed to complement the LARA language and the Kadabra Weaver, as both do not directly provide runtime adaptivity features. The API provides the Java features for adaptivity of parameters and of algorithms, as well as features for runtime profiling and exploration of configurations. This API is integrated with a set of LARA aspects, the Kadabra API, that can be imported and used by other LARA aspects. The main features of the API were developed based on state-of-art approaches. However, if this API does not suffice, LARA allows developers to use a different API or to extend the current one with the necessary features, without compromising the existing work.
Support for Runtime Adaptivity at Software Level

Table 3-I. DSL + API solution for the gathered user stories.

<table>
<thead>
<tr>
<th>As a developer, I want to…</th>
<th>DSL+ API Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>US1 specify application points (locations) and knobs so that I can apply runtime adaptivity rules on those points and to those knobs</td>
<td>LARA (with Kadabra) select statement for join point selection and actions for weaving and code transformations</td>
</tr>
<tr>
<td>US2 specify adaptivity rules and/or automatic search schemes so that I can experiment with and deliver runtime adaptivity applications</td>
<td>Java runtime adaptivity API providing knob/algorithm exploration and autotuning with user-defined exploration schemes</td>
</tr>
<tr>
<td>US3 have a separation of concerns approach so that I don’t need to manually maintain multiple versions of the application code</td>
<td>Strategies defined in the LARA language and the Kadabra Weaving tool to adapt the code</td>
</tr>
<tr>
<td>US4 express reusable and modular runtime adaptivity strategies so that I can possibly apply similar strategies to different applications or to multiple versions of the same application</td>
<td>LARA aspects (using wildcards and inputs/outputs) to produce reusable modular strategies</td>
</tr>
<tr>
<td>US5 express runtime adaptivity regarding application parameters, algorithm choice, and code specialization so that I can approach runtime adaptivity in different contexts and application domains</td>
<td>Kadabra API coupled with the Java API providing runtime adaptivity feature, including knob manipulation, algorithm selection, and runtime code specialization</td>
</tr>
<tr>
<td>US6 have an approach with minimal extensions to the programming language of the application so that it will be easier to learn and faster to use</td>
<td>LARA basis (JavaScript) close to Java programming language. LARA actions accepting Java native code.</td>
</tr>
<tr>
<td>US7 use existing runtime adaptivity and exploration APIs and tools, as well as third party features that aid the development of adaptivity strategies</td>
<td>LARA is extendable with any third party API and tools</td>
</tr>
</tbody>
</table>

The next section discusses the necessary changes to the LARA framework to accomplish the goals of this thesis. The definition of adaptivity strategies may not be straightforward. This is why the following sections also discuss the important concepts on the development of a runtime adaptivity strategy. Relevant execution points are enumerated and the design of a strategy is discussed, including how the adaptation can be triggered (sequential or concurrent) and the runtime metrics to use.

3.3 LARA Framework for Adaptivity and Targeting

Object-Oriented languages

The LARA 2.0 language version (Cardoso, Coutinho, et al., 2012) contained many of the features required for this project. The improvements required for this work were related to modifications over the framework itself, more specifically about how to develop new weaving environments, easy update and aspect decomposition for more reusability. This work improved the LARA language and its framework in the following manner:

- Development of an interpreter (*larai*) able to deal with different weaving environments;
- Development of a weaver generator to develop new weaving environments interfacing with *larai* (the LARA interpreter);
- Mechanisms for aspect composition and API-based strategies for reusability;
- Development of a weaving engine for Java and integration of a Java-to-Java compiler;
- Lara integration with Java and other OO languages overall.
3.3.1 Weaving Framework

The development of new weaving environments for LARA required the complete interpretation of the Aspect-IR generated by the LARA compiler (larac). The LARA interpreter (larai) is an interpreter developed in Java for the Aspect-IR and can be used in different contexts, for instance, weaving process or design-space exploration (DSE) (Cardoso, Diniz, et al., 2013). larai has been developed to simplify the integration of the LARA language in different weaving environments. It executes most of the code defined in a LARA file (e.g., loop and conditional statements), and a weaver being responsible for performing weaving-related tasks. A standalone version of the interpreter has been used for different purposes (Cardoso, Carvalho, et al., 2013): as a scripting language based on the JavaScript syntax, to execute external tools, to coordinate compilation flows (e.g., instrumentation → optimization → compilation → execution → profiling) or to define DSE schemes (Cardoso, Carvalho, et al., 2013).

However, as a standalone tool, the interpreter cannot carry out weaving-related tasks, such as selecting points in the code and applying actions. In order to do so, larai must be connected to a Weaving Engine that is responsible for building an intermediate representation (IR), e.g. an abstract syntax tree (AST), for the target application, select join points, retrieve attribute information, apply actions, and generate the modified code.

To ease development of new weaving environments, larai provides a Java API for Weaving Engines, which works as a bridge between the LARA interpreter and the IR of the target application. This interface, presented in Figure 3.3, reduces the effort needed to develop a new weaver and requires the implementation of abstract classes by the weaver developer:

- **WeaverEngine**: abstract class representing the weaving engine, with methods for connections between the interpreter and the weaving engine.

- **JoinPoint**: the join point abstraction class to symbolize join point instances, providing the required methods for selecting a subsequent join point, retrieving a joint point attribute value or applying an action.

As an example of the level of framework reuse, it is possible to achieve with, consider the attribute-based filters that can be defined in a select statement for each join point. The weaver developer only needs to implement the methods that return each join point type and each join point attribute, being the responsibility of larai to request the required join points, attributes and perform the filtering. Additionally, this API allows larai to provide out-of-the-box, to all weavers, a customized integrated development environment that contains a LARA aspect editor with syntax checker and integrated controls for weaving actions, as well as a language specification guide.
The selection method uses no filter and requests all the join points the weaving engine can obtain from the current pointcut. The filtering is subsequently dealt with by larai. This interface facilitates the development of new weavers. Specifically, it only requires the implementation of the mentioned abstractions for passing information to the interpreter, which performs all the remaining tasks within a LARA aspect (Cardoso, Diniz, et al., 2013). Two examples of weaving engines using this approach are a weaver targeting the MATLAB language (Bispo et al., 2013) and another weaver targeting C/C++ (Pinto et al., 2018).

The LARA Engine includes larac and larai, last one interfaces with a weaver engine, the tool that selects and advices pointcuts. The weaver engine can use any source-to-source compiler to parse and regenerate the target code.

3.3.2 Weaver Generator

Even with the larai API, developing a Weaving Engine from the ground up can still require considerable efforts, mostly due to the fact that each join point in the Language Specification requires a class, each with code that connects the Weaver Engine to the interpreter. Additionally, manually maintaining the Weaver Engine in sync with the Language Specification join points, select relations, attributes and actions are too costly and error-prone. Some of the possible occurring problems are: the names of the join points in the language specification may differ from the handcrafted code, action parameters mismatch, and unimplemented join points and methods for selects, actions and attributes.

Fortunately, a significant part of this effort can be automated. The Language Specification and larai weaver-related interactions have a close association, as all the possible weaving requests are described in the specification. To complement the weaving API provided by larai, the Weaver Generator was developer, which generates the skeleton of a new Weaving Engine from a Language Specification. This generator allows quicker development of new weaving environments, as the developer just needs to implement the abstractions provided by the skeleton.

Although the number of generated classes that have to be extended is similar to the number of classes that had to be manually created, the generator implements all required infrastructure.
Support for Runtime Adaptivity at Software Level

and interfacing code. All join points are generated as abstract classes, and their selects, attributes and actions are added to the class as abstract methods. The abstract class guarantees that the join point implementation conforms to the description in the Language Specification, as it forces the weaver developer to implement the attributes and actions specified. The work left for the developer is essentially focused on writing code for the abstract methods, which would consist in accessing the IR for selection, attribute queries and action purposes, working directly on the chosen IR.

As an example, Figure 3.4 shows a small part of the Kadabra weaver, which targets the Java programming language. To develop the weaver, the Language Specification is given to the Weaver Generator, which generates the abstract classes (see the Interface section). Kadabra extends an existing Java-to-Java compiler, Spoon (Pawlak et al., 2016), which builds the IR for Java code (see the Source-to-Source compiler section). The work of the weaver developer is to implement the Weaving Engine component and the concrete join point classes (see the Weaving Engine section), which bridges the generated interface and Spoon. In the example, this effort corresponds to the development of the concrete classes KadabraEngine, JMethod and JBody, which extend the automatically generated abstract classes.

Figure 3.4. Representations of the code generated by the Weaver Generator (Interface), the code developed by the weaver developer (Weaving Engine), and the code of the source-to-source compiler.

The Weaver Generator allows weavers to be developed incrementally. The Language Specification does not have to be completely defined when the weaver is initially developed. Whenever required, the Language Specification can be upgraded with new join points, attributes and actions and given again to the generator. New classes are added to the Weaving Engine and existing classes are updated with new selections, attributes and actions. Since the Weaver Generator works at the abstract classes level, it does not conflict with the code implemented by the weaver developer.

The left side of Figure 3.5 shows part of the Language Specification for a weaver. The specification is organized in three models, specified as XML files: join point hierarchy, artifacts and actions. The join point hierarchy defines available join points and how to select them. The example declares a join point named method, which is able to select its parameters (which are
Support for Runtime Adaptivity at Software Level

var join points) and body (the declaration of the var and body join points is omitted in Figure 3.5). Aliases can be used when one intends to give a specific meaning to the type of join point wanted to be selected. For instance, in this example, the alias param is used to specify that one wants the parameters of the code, but they are internally join points of type var. The artifacts model defines the attributes of a join point, such as method name and return type. The actions model defines the actions that can be applied to the join points.

Based on the Language Specification, the generator creates the abstract classes, and the weaver developer implements the concrete classes. The class Method shows the bridge between the LARA interpreter and the source-to-source compiler as it uses a node from the AST of the compiler to get values of attributes or to apply the clone action.

An advantage of such approach is that the Language Specification does not need to provide a one-to-one mapping to the target language or selected IR. Instead, it needs only to specify which parts are considered points of interest and which attributes they have.

3.3.3 Aspect Composition for Reusability and Multiple Target Languages

This section describes the developed features and techniques to achieve modular and composable aspects, which in turn lead to more reusable aspects that may even target different programming languages. The techniques include LARA libraries, bundles of user code, and defining the join point model in a way that increases compatibility between languages, while maintaining most language features intact. The benefits of these techniques can be harnessed across several levels, depending on where they can be implemented: at the LARA framework level (becomes available to all weavers), at the weaver developer level (becomes available to all weaver users) or at the weaver user level.

LARA aspect definitions (or aspectdef) are treated as independent modular units which can be called from within other aspectdefs. They can be parameterized and are generally treated as functions (they are callable, execute a set of instructions, including advices, and return output values). This helps to separate sub-concerns of larger strategies, leaving individual aspectdefs...
to perform well-defined tasks, improving the modularity of the entire program and maintainability of aspect code. LARA aspects can also import other compilation units with one or more aspectdefs. This means that the reused aspect code does not need to be in the same aspect compilation unit. This also means that users may develop their own aspect libraries, reuse them and distribute them.

For instance, one might define an aspect with clearly defined pointcuts and then use several aspect definitions to add different behavior to the application, effectively promoting reuse at the pointcut level. Conversely, it is also possible to define additional behavior for a common concern once, and then use it for multiple different pointcuts, promoting reuse at the advice level.

One can have JavaScript-based libraries, called from anywhere inside an aspectdef that are useful for simple tasks, such as code generation. The benefits of using code generation libraries are threefold. First, there is the obvious reuse of code. A developer can write the code generation library, test it and use or provide it to other developers. Secondly, an important benefit is that aspect developers may need to write less native code. In turn, this likely speeds up aspect writing and make it less error-prone, as well as also improve intelligibility. And in third, the developers may write their own versions of such a library in a way that the library is able to target different languages. The last one would transform this into a multilanguage library.

Figure 3.6 shows an example of an aspect for timing the execution of a function call. Figure 3.6 (a) shows an aspect using native Java code to get the current time before and after function calls and reporting the difference of those two values as the elapsed time. Figure 3.6 (b) shows an aspect that uses an API to measure function calls in a more abstract way. First, this is a reusable aspect that can be called from other strategies and target specific calls by using the fName parameter. Second, it did not require the aspect developer to write any native line of code. Last, this aspect can be used over any target programming language (unlike the one in Figure 3.6 (a)) that has the concept of function calls, providing that a weaver exists for that target language.
Support for Runtime Adaptivity at Software Level

```java
aspectdef TimeCalls
  input fName = ".+" end
  // Time calls to fName
  select call{name==fName} end
  apply
    // Get time before the function call
    $call.insert before "long start = System.currentTimeMillis();"
    // Get time after the function call and report the difference
    $call.insert after %{
      long time = System.currentTimeMillis() - start;
      System.out.println("Time: " + time + "ms");
    }%;
end
end
```

(a) Lara aspect with specific Java code to get the current milliseconds

```java
import Timer;
aspectdef TimeCalls
  input fName = ".+" end
  // Create microseconds timer that prints information to the console
  var timer = new Timer("MICROSECONDS");
  // Time calls to fName
  select call{name==fName} end
  apply
    timer.time($call, "Time: ");
end
end
```

(b) Lara aspect using an API to measure the call in microseconds

Figure 3.6. Timing calls to functions which name matches fName using (a) native code and (b) a LARA API.

Most uses of LARA require more complex operations than this simple code injection operation. There is added complexity when defining pointcuts and, generally, the advice code to be weaved is considerably more complex than the previous example. However, even at this level, one can still find aspects repeating the same concerns, meaning he/she could likely develop abstractions and include them in a library. Aspect libraries are used when there are common tasks that query and modify the target code. Examples include monitoring, such as timing parts of the program (e.g., function and loop execution), logging activities, such as error conditions and function calls and complex code manipulation techniques.

Aspect library bundles exist so that end users can also develop their multilanguage aspect libraries and distribute them. This is a compromise between the concepts of Interface and Aspect Multiversioning, since a user provides a library interface with corresponding implementations, but also marks each version as targeting a specific weaver. The weaver then reads all the needed files and imports the correct ones. This process is transparent to the end-user that just intends to use a library distributed as a bundle. As these libraries are developed and provided by users, the interface to the library can be different depending on the language.

If besides the library APIs, weavers also share the part of the Language Specification used in an aspect (e.g., the join point call is usually used for function/method invocation), the aspect can be fully compatible between weavers, even in cases where the target language is not the same (see, e.g., (Pinto et al., 2018)).

When designing a new weaver, one of the first steps is the definition of the Language Specification, which states the points in the code that may be selected, their attributes and the
actions that can be applied over them. Internally, when developing weavers, one needs to be sure there is some overlap between the join points and attributes available for all weavers.

Besides sharing join points and attributes among weavers, two more techniques can be considered, generic weaver actions and join point aliases. Generic weaver actions can be defined if several weavers conform to the same standard for actions and provide the same semantics, even if such actions are considered weaver specific. In certain cases, there are points of interest in the code that are similar between languages, but that can have different names due to nature, history or conventions of the language (e.g., function in C vs method in Java). To increase compatibility between weavers, when specifying a language, the weaver developer can use join point aliases, which allows referring to the same join point using different names. For instance, in Kadabra, the Java weaver depicted in section 5.2, function is an alias for method, which means one can capture methods with any of the select statements depicted in Figure 3.7.

```
1 select method end
2 // ... advice code here
3
4 select function end
5 // ... advice code here
```

Figure 3.7 Equivalent select statements since function is an alias for method in the Kadabra Weaver.

With join point aliases, instead of forcing a single denomination to all languages, weavers can use their conventional denomination and still have compatibility with more generic aspects.

## 3.4 Execution Points

To define an adaptation strategy, one has to know the adaptation goal and how to accomplish that goal. The objective of the adaptation is usually related to a non-functional requirement for the application. Decreasing execution time and energy consumption are two examples of adaptation goals. How these goals are achieved is what the developer describes in adaptation strategies. Hotspot optimizations for execution time and method offloading for energy savings (Kumar et al., 2012, Zhang et al., 2012) are two possible solutions for the previous goals. Depending on the accessibility over the application and/or the system, one may achieve adaptation by changing specific program instructions/statements and by using system parameters and operations. How the adaptation occurs can be accomplished with different forms, from the adjustment of a knob (Hoffmann et al., 2011), e.g. program variables or system-related parameters, to the reformulation of the executing code, such as algorithm swap (Wernsing and Stitt, 2010) and runtime optimizations (Voss and Eigemann, 2001).

Adapting the application at runtime requires special reasoning about when and where to apply changes is fundamental in order to maintain program correctness and/or acceptable resulting application behavior. The proposed approach considers that runtime strategies have to deal with at least four application execution points: adaptation, trigger, update and measurement.

Adaptation defines the actual point in the code where the program is adapted, i.e., where the dynamic strategy applies the changes. A strategy may be defined resorting to only the adaptation point. This means that this point triggers the adaptation, immediately executes the strategy code and updates the program. However, many strategies most likely do not perform the
adaptation execution where the adaptation occurs, but in a more strategical location in the program. Let us consider the synthetic example of Figure 3.8, which defines a hotspot, composed by nested loops and a specific (inner) statement, the adaptation target, and hypothetical artifacts that are going to be used to select an algorithm to execute, which are not updated inside the hotspot. In this case, the adaptation should be triggered only once, right before the outermost loop is executed (i.e. before the for loop in line 3).

![Figure 3.8. A synthetic method consisted of three nested loops invoking a single method.](image)

With this description one can decide, instead of executing the adaptation during all the loops iterations, to execute the adaptation before the hotspot itself, e.g., before the outermost loop statement. Additionally, using this point alone limits the adaptation opportunities, e.g. code retransformation in the same location the program is executing does not reflect in the current method execution, only in the next execution.

The trigger point determines where the strategy is executed. When the execution crosscuts the trigger point it executes the strategy to adapt the target code. At this point adaptation, event-based conditions are defined, as it may not be intended to execute the strategy every time the execution intersects the trigger point. One could analyze the program execution before making any adaptation decision, or explore different parameters every \(N\) seconds. These events take different forms and are defined by the strategy developer. For instance, a periodic event may be used when it is intended to experiment with multiple versions of code, where each version is monitored a number of times or during a period of time. Some examples of events are:

- Arbitrary conditional expressions: toAdapt == true && newVersion != null
- Runtime event: field update, the battery level is below/above a certain threshold
- Periodic: every \(N\) calls to a function, every \(T\) milliseconds

Figure 3.9 depicts an example in which the call to method `foo` is adapted every 10 executions of method `bar`. Since the adaptation will not use any artifacts of the nested loops the strategy is defined at the beginning of the method (lines 2 to 4).
Support for Runtime Adaptivity at Software Level

The update point defines a location in the code where the application can be updated with the required changes, i.e., the location to replace a method or to redefine a parameter. Following the previous example, consider that the adaptation strategy was triggered and is executing concurrently to the hotspot execution to produce new code/values for the target statement. If these new changes are applied during the hotspot execution, the program could deal with unexpected behavior between loop iterations. The safest place to replace the target method would be before or after the loop is executed, where the next hotspot execution would reckon the adaptation. While the runtime strategy can be applied in parallel or sequentially to the application, the application update is always sequential, as this update point is advising the program that, from that point on, the application is updated.

Figure 3.9. Adapting call to foo every 10 executions of method bar.

```java
bar (float[][][] array) {
    if (barExecs >= 10) { //adaptation trigger
        //adapt foo & reset counter
    } else barExecs++;
    ...
    for (int i = 0; i < limitI; i++) {
        for (int j = 0; j < limitJ; j++) {
            for (int k = 0; k < limitK; k++) {
                sum += foo(array, i, j, k); //target statement
            }
        }
    }
}
```

Figure 3.10. Call to foo is adapted with new versions. The definition of new sampling versions is concurrent to the execution.

```java
bar (float[][][] array) {
    ...
    if (newVersion) {
        ... //update call to foo
    }
    for (int i = 0; i < limitI; i++) {
        for (int j = 0; j < limitJ; j++) {
            for (int k = 0; k < limitK; k++) {
                sum += foo(array, i, j, k); //target statement
            }
        }
    }
    trigger(); //concurrently adapt the application
}
```

The measurement/feedback point is the point, or code section, that provides execution feedback. This can be by default the same location as the adaptation point or in a more strategic location. Doing measurements at runtime may impose some unacceptable overhead and so it is better to inspect the best location for the measurement point. Continuing the previous example, Figure 3.11 shows the example considering that the best location to do the measurements is around the outermost loop (lines 3 to 6, 14 and 15). This avoids the overhead of multiple
Support for Runtime Adaptyev at Software Level

assignments to the foo variable, multiple calls to start/stop measurements and multiple feedback
calls to the adapter.

```c
bar (float[][][] array) {
    ...
    if (newVersion) {
        ... //update call to foo
    }
    startMeasure();
    for (int i = 0; i < limitI; i++) {
        for (int j = 0; j < limitJ; j++) {
            for (int k = 0; k < limitK; k++) {
                sum += foo(array, i, j, k);  //target statement
            }
        }
    }
    fooTime = stopMeasure();
    trigger(fooTime); //use a strategy considering the execution time of the
    //current "foo" to decide the following adaptation
}
```

Figure 3.11. Call to foo is adapted with new versions but measurements are done around the outer most loop.

The four execution points described above are fully supported by the approach presented in this thesis throughout LARA and Kadabra. The approach has the capability to support strategies that automate the selection of those execution points.

3.5 Design of Strategies

Programs that are target of an adaptation strategy are usually not prepared to deal with such changes. When prepared for the adaptation means that these concerns are already contained within the main concern of the program. The key feature of AOP is the separation of cross-cutting concerns from the target code (Kiczales, 1996). The proposed approach preserves this idea by keeping the adaptation strategies apart from the code, maintaining program independency from these concerns. In order to make programs “aware” of the adaptations, one must first prepare the program to accept adaptation requests. This means that the strategies development may require some kind of program restructuring in order to define adaptation mechanisms.

The presented approach considers that the specification of adaptation strategies consists of a two-phased process: code restructuring and adaptation description. The first phase defines static modifications on the program to comprise the runtime strategies while the second phase specifies the strategy by means of the adjustments of the first phase. The following sections discuss these phases and how they coexist within a strategy.

3.5.1 Compile-time Code Restructuring

Compile-time code restructuring is specially important for applications under development, or already developed, where (original) code maintainability is important and to avoid unexpected behavior when the code is manually modified. During the first phase, it is intended to prepare the application for the adaptation strategy. In this phase, one applies compile-time transformations and adds new functionalities to be used by the adaptation definition phase. This is the typical
Support for Runtime Adaptivity at Software Level

source-to-source weaving environment, defining code transformations to optimize the program with static and/or profiling information. Code transformations expose specific code locations, and parameters, to provide runtime information retrieval and to allow online transformations.

Figure 3.12 shows examples of code transformations and their use in an adaptation strategy. New variables allow to store strategy information, such as monitors and exploring values, as the interface between the target program and the strategy, to be used as a software knob. Some code transformations, such as loop unrolling and loop interchange, allow pre-optimization of the code, while others may simply rearrange the code for other goals (e.g., functional interface extraction for method call replacement). Another way of preparing the code is to inject library functionalities in the program, for instance, to execute the adaptation concurrently or to use a specific API for measuring energy consumption.

Figure 3.12. Examples of code transformation and injection to prepare a program for adaptation.

The adaptation description might be entirely dependent on this phase to be able to access certain points of interest in the code and add the runtime adaptivity. Take the example of a call to the static matrix multiplication method depicted in Figure 3.13(a). If there are different implementations of this matrix multiplication and one intends to program a strategy that explores the different algorithms instead of the static function call, then this phase shall first restructure the function call in a way that it can be replaceable at runtime. Figure 3.13(b) shows one way to do this, which adds a variable to store the method to use for the multiplication and doing so allows the strategy to adapt this variable, i.e., to reassign its value so a different method is called.

```
1 ... 
2 m3 = MatrixUtils.mult(m1,m2); 
3 ...
```

(a) Original static call to mult

```
1 IMult matrixMult = MatrixUtils::mult; 
2 ... 
3 //code adapting matrixMult 
4 ... 
5 m3 = matrixMult(m1,m2); 
6 ...
```

(b) Adapted version that allows runtime adaptivity

Figure 3.13. A call to (a) a static method named mult that pertains to a class named MatrixUtils and (b) the adapted code to allow runtime adaptivity.
Support for Runtime Adapivity at Software Level

This phase may advise any of the execution points, depending on the type of strategy used. However, the most advised execution point is usually the adaptation point, i.e., the actual point that dynamically changes.

3.5.2 Adaptation Description

At this point, the runtime adaptation can be defined, reckoning the readjustment done by the previous step. The code to be weaved in this stage reflects in the dynamic adaptation. This stage makes use of the execution points to deliver runtime adaptivity. Figure 3.14 shows the flow of an application with runtime adaptivity and how the execution points affect the execution. The code must be instrumented on the execution points with their corresponding purpose. The adapter shown in Figure 3.14 is a simplified view of the work it does. An actual adapter might have much more work to be done. For instance, an adapter may be responsible to manage different search schemes, to dynamically generate algorithms and/or to respond accordingly to the adaptation requests.

The first step is the adaptation trigger (1), the location in which, based on runtime information, one decides about the addition of adaptation and, if so, request the adaptation to the adapter. Based on the information given at this moment, and previous information gathered, the adapter selects (or generates) a configuration to use (2). This configuration, however, is only applied in the main program when the update point is reached (3). At this time the adaptation point is updated with the configuration previously chosen by the adapter (4). The measurement point, after its execution, feedbacks to the adapter the performance of that chosen configuration (5).

Figure 3.14. The usual flow of an application with runtime adaptivity.

The proposed approach provides adaptation at three levels, namely: software knobs, algorithm selection, and program specialization. These levels are used for several purposes and can be combined to achieve multiple goals in a strategy. All of these functionalities are available in the Kadabra API and can be fully controlled by LARA strategies. The type of adaptation intends to weave in the application depends on the targeted adaptation point. Philip et al. (McKinley et al., 2004) depict two general approaches for software adaptation: parameter adaptation and compositional adaptation.
With parameter adaptation, one can manipulate knobs to modify the program behavior. This makes use of exposed variables/fields or a system flag that can be reassigned with a different value. It can even be more complex and consider multiple parameters in the same adaptivity strategy. When the correct value for each possible case is known at compile-time then an adaptation strategy can be defined by adding adaptivity code that switches the value of the parameter according to the current situation.

When the best configuration is unknown it is necessary an approach that allows the exploration of parameter configurations. The developed API provides mechanisms to define search schemes that explore different values for the target parameter. When dealing with multiple parameters it is also possible to have combinations of search schemes. Several types of search schemes are available and, if none suffices the developer needs, the approach allows user-defined search schemes.

However, the support described does not allow one to add and try different algorithms. Compositional adaptation allows one to change algorithms and the structure of the components of the system. This flexibility supports more than simple tuning of parameters and enables dynamic recomposition. Having multiple algorithms for the same objective, an adapter may be defined with a strategy that selects the best algorithm for a given scenario. Furthermore, the proposed approach allows the definition of adaptivity strategies that use algorithms associated with knobs and the runtime generation of algorithms with a template-based approach.

Similar to parameters adaptation, it is possible to design strategies that explore the algorithms, pre-compiled and generated at runtime. For instance, it is possible to define strategies that use chained search schemes to randomize the algorithms exploration and an inner search scheme providing configurations for a knob associated with an algorithm.

### 3.6 Sequential and Concurrent Adaptation

By dynamically adapting an application, one is clearly adding some execution overhead, not existent in the original version. For instance, a runtime specialization event requires (at least) time for the code generation and the redefinition of the target classes. Figure 3.15 illustrates possible adaptation schemes that may occur with the proposed approach. These schemes intend to show how the adaptation overhead (labeled as adapt/launch) may influence the execution time.

The first scheme (original) is the execution of the application without any adaptation. The second scheme (sequential) depicts a sequential adaptation. This means that the application halts its execution to perform the adaptation and then resume its execution with the adapted code. There are two ways of using this scheme: to adapt the application when certain information is obtained, and then a re-adaptation is performed if the information changes, or to explore different code versions periodically. The third scheme (concurrent) is a multi-threaded version in which the adaptation is done concurrently to the application’s execution, not requiring for the application to wait for a new version. This approach can be advantageous in certain situations, where the analysis and generation can be performed parallel to the execution. However, the approach can be discarded when true parallel execution is not possible.
The different schemes may be advantageous in different situations. Hence, the decision of the best approach is shared between the weaver and the developer. The developer is able to decide which type of scheme is using (directly with the programmed strategy), while the API verifies if the system has hardware support for parallel execution of threads.

Figure 3.16 shows an example of a typical sequential adaptation when exploring different configurations. During the program execution, every time that a configuration is requested (1), the adapter verifies if it is still in sampling mode, i.e., analyzing configurations, and if not returns the best configuration found. When in sampling mode, the adapter analyzes the current measurements and if the sampling period for the current configuration has not finished, i.e. the number of sampled iterations less than the predefined limit, then returns the current configuration being sampled. When the sampling period has finished, then the adapter analyzes (2) and updates the best configuration based on the sampled configuration (3) and generates a new configuration if available (4), or stops the sampling mode and returns the best configuration. The application continues with the provided configuration and measures the execution of this configuration (5) and feedbacks to the adapter (6).

In this scenario, the adaptation adds overhead in the execution for getting the configuration, in the measurement of the execution and while updating measurements. Getting a configuration may add overhead due to the possible analysis of a configuration and the generation of new configurations.
Figure 3.16. Example of a sequential adaptation that decides between running the best configuration or explore and sample configurations provided by a search scheme. A version is executed “LIMIT” times and the adapter updates the reference to the best configuration according to the sampling measurements.

Figure 3.17 shows an example of a concurrent adaptation scheme. The measurements feedback and new configurations generation are concurrent to the program execution and only the request of a configuration and the measurements are sequential with the main program. The program starts by requesting a configuration before the execution of the hotspot (1). A concurrently accessed first-in-first-out (FIFO) channel is used to store configurations that still were not sampled and a field is used to store the best configuration so far. If the channel contains a configuration then that configuration is executed, otherwise, the best is used. This is the only sequential part of the adaptation while requesting a configuration. The program resumes its execution and measures the target code (2). This measurement is then fed to the measurements concurrent channel (3) and the program continues to execute.

The adapter is concurrently waiting for measurements to evaluate the current configuration (4 and 5). When the measurements period is finished, that configuration is analyzed (6) to update the best configuration (7) and a new configuration is generated (8) and fed to the configurations channel (9).

This approach is important when the adaptivity search is computationally intensive. For instance, code generation is a heavy task that may be better to be executed in parallel to the execution of the application. The overhead in the program is then reduced as usually most of the computation is done concurrently. The overhead on the program execution would be most likely due to synchronization, messages to the adapter and decisions about whether to use or not the adapter.
Support for Runtime Adaptivity at Software Level

![Diagram showing concurrent adaptation](image)

Figure 3.17. Example of a concurrent adaptation. The actual program and the adapter communicate via concurrent channels, allowing both to be executed laterally. The “offer” method allows asynchronous insertion in the channel, while the getters are synchronized methods.

### 3.7 Goals and Metrics for Runtime Evaluation

While adapting an application based on runtime exploration, it is necessary to define the goals of that exploration. The goal of an exploration will depend on the developer intentions and is based on the score of a given configuration and on the criteria used to compare scores.

A score is a measurement given to each explored configuration and is always associated with the type of goal. For instance, if one intends to measure execution time, then the score may be given in milliseconds or seconds. Runtime measurements are used to evaluate the configurations used during runtime adaptivity. Usual measurements include execution time, energy consumption, memory usage, accuracy and any other metrics a developer might need to use. The developer can also use other types of metrics using APIs that provide those metrics. For instance, one might add an API that can access the temperature of the system devices and other types of data, or even provide analysis over the source code to retrieve the complexity of an algorithm.

The criteria compare scores and decide which one is considered the best value. If the developer goal is to reduce execution time or to save energy then the criteria is to have the value which score delivers the lowest execution time or lowest energy consumption, respectively. If one intends to find the most accurate output, then the criterion is to find the value which score delivers the lowest error possible.

The advantage of this approach is that any measurements and criteria can be used to create adaptivity goals. It is possible to define goals that combine different scores and criteria and define an exploration that tries to find the value with the lowest execution time score but keeping a minimum accuracy score. Another example is a multiobjective strategy that uses execution time and energy consumption as metrics and defines the adaptivity goal to reduce energy consumption but not to delay execution time by a certain threshold. The proposed approach can support multiobjective schemes and goal definitions that are common in runtime autotuning approaches, such as the approach presented by Silvano et al. (2016).
3.8 Summary

This chapter started by discussing the requirements of runtime adaptivity strategies and by analyzing a set of user stories collected from user information discussed in a number of projects involving both academic and industrial partners.

Based on those requirements, the chapter presented the proposed approach using a DSL+Weaver+API, followed by the improvements done in the LARA framework to support the definition of runtime adaptivity strategies and the requirements for the specification of runtime adaptivity, i.e., the important concepts necessary to efficiently specify strategies delivering runtime adaptivity. These important steps include the execution points one has to consider to add adaptivity in the application, how the code should be restructured to comprise adaptivity and how a developer decides what to do at runtime in terms of runtime adaptivity with the contextual information that can be used. It is also discussed sequential adaptation versus concurrent adaptation and how to use metrics and goals to evaluate the runtime adaptivity.
Support for Runtime Adaptivity at Software Level
Designing and Mapping Runtime Adaptivity Strategies

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Software and Optimization Knobs</td>
<td>78</td>
</tr>
<tr>
<td>4.2 Algorithm Selection</td>
<td>82</td>
</tr>
<tr>
<td>4.3 Runtime Generation and Specialization</td>
<td>88</td>
</tr>
<tr>
<td>4.4 Search Schemes</td>
<td>94</td>
</tr>
<tr>
<td>4.5 Autotuning</td>
<td>97</td>
</tr>
<tr>
<td>4.6 Summary</td>
<td>101</td>
</tr>
</tbody>
</table>
Designing and Mapping Runtime Adaptivity Strategies

This chapter presents how adaptivity strategies are specified and describes what it is necessary to do depending on the runtime adaptivity one intends to use. It is organized by the type of adaptivity the proposed approach may provide. The complete design of runtime strategies is presented, from the compile-time restructuring to the runtime adaptivity scheme. The strategy algorithms are presented to show how a developer may design strategies by means of a DSL providing runtime adaptivity.

4.1 Software and Optimization Knobs

Tuning of parameters, also known as knobs (Hoffmann et al., 2011) or control points (Dooley and Kale, 2008), is a simple yet powerful adaptive mechanism for autotuning. This approach is based on exposing and controlling application and system parameters to be dynamically tuned. Here, three types of knobs that can be used in an adaptation strategy are considered, namely: software, e.g., fields and variables; system, such as Java Virtual Machine (JVM) flags (Jayasena et al., 2015), and code optimizations. More examples include knobs for setting up application options and disconnect application or system features and devices.

Software knobs are the simplest way of having runtime adaptivity in an application, as these are usually easy to access and to reassign. They can be local variables or fields. Depending on the information required to make the adaptation, one may have to reallocate the target variable, or even an expression, to a field or to redefine the assign meant to a variable.

Consider the example of exposing and tuning the number of neighbors, i.e., k, of a kNN classifier (Mucherino et al., 2009), where the tuning criteria are known at compile-time. In this example, the kNN classifier is a Java class that contains a field k and setK and classify methods. This is an example of a software parameter that can easily be adapted without any code restructure needed. Figure 4.1 shows a strategy for tuning k based on that tuning criteria. In this example, the developer knows that the adaptation point is the k field and the strategy considers the trigger and update point the same. The feedback point is ignored in this strategy as the developer does not want any feedback from the application. The developer adapts k value before calls to the classify method based on criteria fetched from a selected expression. Then, the developer specifies the code to be inserted before each call, a switch that changes the k value based on that criteria. The Java code to inject might even be an expression implementing an equation instead of a switch. The main point is that the adaptation code to inject is user-defined.

```java
select calls to method classify from class kNN
select expression from ... as kCriteria
foreach call using kcriteria
execute before call
  switch(kCriteria){ //pure Java code
    case crit1: kNN.setK(3); break;
    case crit2: kNN.setK(5); break;
    case crit3: kNN.setK(7); break;
    ...
  }
end execute
end foreach
```

Figure 4.1. Example of a strategy that tunes the number of neighbors of kNN based on runtime criteria.
Designing and Mapping Runtime Adaptivity Strategies

This would be the most basic adaptivity strategy one could define for a software parameter. Applying adaptivity to optimization knobs is very similar. Some compiler optimizations have ways or exposable parameters that allows the tuning for increasing performance. Loop tiling, for instance, is a transformation that tries to improve data locality (Xue, 2012) and uses a specific size to control the tiling. Figure 4.2 shows an example of a loop advised with a loop tiling transformation using a fixed block size.

```java
for (int iBlock = 0; iBlock < iLength; iBlock += 32) {
    for (int i = iBlock; i < min(iLength, iBlock + 32)); i++ ) {
        ...
    }
}
```

Figure 4.2. Loop tiling with a fixed block size of 32.

The block size depends on several factors, including the size of data being processed in the loop and the processing unit cache size. A runtime adaptivity strategy may expose the block tile size as a variable, allowing to easily tune its size within an adaptation strategy, i.e., an optimization knob (Tavarageri et al., 2010). This may even be automatically applied if the loop tiling transformation does expose, or have the meanings to expose, the block size.

Figure 4.3 shows a strategy for adapting a parameter by means of an exposed block of a tiled loop. The example starts by preparing the target loop by extracting the static block size to a variable. Then, the strategy defines that prior to the outermost loop execution, the block size should be assigned with a half value of the L2 cache size. To get the L2 cache a library having access to CPU information is added. This is an example in which code restructuring was necessary to add the API to be used at runtime and to expose an expression.

```java
import API CpuInfo
select loop //both loops
select expressions with 32
    foreach expression
        extract expression to variable named block
    end foreach
select loop if outermost
    execute before loop
        block = CpuInfo.getL2CacheSize();
    end execute
```

Figure 4.3. A high-level strategy that exposes the block size of a tiled loop and dynamically redefines its value with half the size of L2 cache, obtained using an API that retrieves CPU information at runtime.

These approaches specify adaptivity only based on static and profiling information. Meaning that although all decisions are made at runtime, the tuning criteria are based on information gathered by the developer and are directly defined in the adaptation strategy. When dealing with unpredictable behavior and different systems the offline profiling becomes more a starting reference than a adaptivity criteria. Furthermore, offline-based decisions become impossible in situations where the adaptivity criteria are unknown. For the loop tiling transformation example, the ideal block size might not always be the same, even if the application is executing in the same environment or in a different one.

This demands an approach that allows one to define strategies that based on a given scheme and a goal, search for the value which achieves that goal. Searching for a value means that there is a range of values the knob can take but the appropriate value to use at runtime is unknown. Ranges can vary from discrete values, e.g. true/false or -1/0/1 values, to continuous and large
Designing and Mapping Runtime Adaptivity Strategies

ranges, e.g. integers or floats. A range can be too large for the exploration hence the offline profiling can be used to reduce the size of that range. For instance, one can define a range with only positive integers and limit to a max size of N. More information about how to define ranges and search schemes is presented in section 4.4.

Designing a strategy that provides this type of adaptation is not trivial and the code restructuring may need to add new components to the application. First of all, a few variables are necessary to keep track of the range of values, the explored values (to avoid repetition), the current best value and its score, the current sampled value, the current score, the number of samples performed on the current score, the measurement code, and a boolean to turn exploration on and off. Then, some code logic is needed to provide new values, inside the specified range, to be explored, and measure the program execution to provide a score to that value. After all, or a set of, possible values are explored the application can then execute using the version with the best score.

Figure 4.4 shows the same code as in Figure 4.2 but with the exposed loop tiling block and the adaptation strategy code. Lines 1 to 9 introduce new fields to control the exploration, including a timer to be used to measure the execution time of the outermost loop. Lines 12 to 27 introduces the exploration code. The first condition verifies if the exploration is executing, then it verifies if there are enough samples to compare the current sampled block size with the best block size. If not continue normal execution. If so, first update the best block size by comparing the scores of the current block and the best block. Then use double of the current value as the new block size to sample. If the double surpasses the upper limit of the range, then stop the exploration and always use the best block size. Lines 29, 35, 36 and 37 show the measurement of the outermost loop to update the current value.
Designing and Mapping Runtime Adaptivity Strategies

```java
int lowerLimit = 8;
int upperLimit = 2048;
int bestBlock;
Score bestScore;
int block = lowerLimit;
Score iScore;
int numSamples = 0;
boolean explore = true;
Timer timer = new Timer();
...
if (explore) { //trigger and update point
  if (numSamples > LIMIT) {
    if (iScore < bestScore) { //criteria
      bestBlock = block;
      bestScore = iScore;
    }
    block = block * 2; //new value to explore
    if (block > upperLimit) { //exploration has finished
      block = bestBlock;
      explore = false;
    } else {
      iScore = new Score();
      numSamples = 0;
    }
  }
  timer.start(); //measurement point
  for (int iBlock = 0; iBlock < iLength; iBlock += block) {
    for (int i = iBlock; i < min(iLength, iBlock + block)); i++) {
      ...
    }
  }
  timer.stop();
  iScore.add(timer.getTime());
  numSamples++;
}
```

Figure 4.4. Loop tiling combined with an adaptation strategy that searches for the best block size (from 8 to 2048) based on the execution time of the outermost loop. The search uses increments of powers of 2 to reduce the exploration size.

The strategy becomes more complex if the scheme of the exploration is not as straightforward as exploring all values as depicted in Figure 4.4. For instance, if one intends to start the exploration from a specific value and explore the surrounding values until no better version is found, then the exploration code increases significantly. Figure 4.5 shows the changes required in the code of Figure 4.4 to have this exploration scheme. Two additional fields are now necessary: a queue to control the values to explore and a set with the already explored values. In addition, the adaptation code augments considerably. The neighbors of a value are added to the queue only if that value improved the current best score if they were not previously explored and they are within the range limits (lines 10 to 24). The next block size to explore is pulled from the queue. If the queue is empty, then there are no more values to explore and the exploration ends.
Designing and Mapping Runtime Adaptyivity Strategies

```java
int lowerLimit = 8;
int upperLimit = 2048;
int block = 128;
Set<Integer> explored;
Queue<Integer> toExplore;
...

if(explore)  { //trigger and update point
  if(numsSamples > LIMIT) {
    if(iScore < bestScore) { //criteria
      bestBlock = block;
      bestScore = iScore;
      lowerBlock = block/2;
      upperBlock = block*2;
      if(lowerBlock > lowerLimit && !toExplore.contains(lowerBlock) && !explored.contains(lowerBlock)) {
        toExplore.add(lowerBlock);
        explored.add(lowerBlock);
      }
      if(upperBlock > upperLimit && !toExplore.contains(upperBlock) && !explored.contains(upperBlock)) {
        toExplore.add(upperBlock);
        explored.add(upperBlock);
      }
    }
  }
  if(toExplore.isEmpty()) {
    block = bestBlock;
    explore = false;
  } else {
    block = toExplore.poll(); //new value to explore
    iScore = new Score();
    numSamples = 0;
  }
}
```

Figure 4.5. Search scheme for the best block size that starts with the value 128 and when the sampling results in improvement it continues the search with the direct neighbors.

This approach is more useful when the developer wants complete control over the adaptation code. The exploration is easier if the approach provides at least access to basic exploration functionality, such as the automatic introduction of exploration control variables and mechanisms to provide values based on a search scheme. An automated and simpler approach for the exploration of knob values is depicted in section 4.4, requiring less effort from the developer but limiting his/her decisions on the weaved code of the adaptation.

The two simple examples presented above strongly shows the typical level of modifications required to aid application with runtime autotuning /adaptivity. Other schemes may require even more complex code (see, e.g., (van Laarhoven and Aarts, 1987, Dooley and Kale, 2008))

### 4.2 Algorithm Selection

The selection of the best algorithm is an approach to achieve the best performance for specific execution scenarios (see, e.g., (Kotthoff, 2016, Wagner et al., 2018)). Having multiple algorithms with the same objective, each with advantages for specific circumstances allows a selective approach based on runtime information. For example, sorting is an effective example of an
operation for which numerous algorithms exist, each with advantages and disadvantages, according, e.g., to the input values/range and target computer architectures.

An approach for algorithm selection is to replace its invocation with conditional switching among the algorithms. This approach is feasible when the best algorithm is statically known or when the best selection can be based on the use of runtime properties. For scenarios where the best versions are unknown, a dynamic approach is required. The adaptation at this level requires that the code must be prepared at compile-time to deal with the algorithm selection. This means that invocations to the target method must be defined in a way that it can be easily changed. One way to do this is to extract the method invocation to a variable/field and invoking this field instead. This way, it is possible to deal with method selection in a similar way to a software knob, i.e., the algorithm selection can be seen as tuning of a parameter.

Figure 4.6 shows a strategy that follows exactly this approach. The first step for restructuring the code is to access the method one intends to adapt (line 1) and introduce a new interface in the application and a method inside that interface (lines 2 and 3). The interface uses the same signature as that method. If such an interface already exists in the code, then this first step is not necessary. After that, all calls to that target method are extracted as fields and replaced with invocations to that same field (lines 6 and 7). Having the code restructured, it is now possible to change the algorithm to use at runtime. The developer selects the update point and all relevant runtime information necessary to decide which version to execute (lines 9 to 11), and then inserts in the update point the code that selects the proper algorithm to use (lines 12 to 19). This approach makes the second part of the algorithm selection very similar to software knobs.

```
1 select method with name == targetName
2 introduce interface in app (name= "FuncInterface")
3 introduce method in interface (name= targetName, signature= method.signature)
4
5 select call with name == targetName //all calls in the app
6 extract call to variable of type interface named func
7 replace call with func.invoke(args= call.args)
8
9 select func //the adaptation point
10 select updatePoint //user-defined update point
11 select relevantInfo //necessary runtime information
12 execute before updatePoint
13 switch(relevantInfo){
14     case scenario1: func = alg1; break;
15     case scenario2: func = alg2; break;
16     case scenario3: func = alg3; break;
17     default: func = defaultCall;
18 }
19 end execute
```

Figure 4.6. A strategy that extracts a functional interface based on a method signature and extracts a method invocation to a variable. The algorithm to invoke is then dynamically adapted.

This approach allows one to have different ways to switch between the different algorithms, whether by the conditional branching or by a collection-based approach (e.g., list or map). The approach considers three possible switching types: conditional branching, collections and bytecodes replacement via code transformation.

Conditional Branching is based on a conditional decision of which algorithm to execute. This can be achieved by means of if-then-else and switch statements, as shown in Figure 4.6. In terms of computation, this approach is expected to have the lowest overhead of the three
techniques and allows freedom on the criteria expression. However, this comes with some limitations. Depending on the number of versions it may pollute the code too much, increase code complexity and the code size. It cannot dynamically add new scenarios, and, consequently, new algorithms or algorithm derivations.

Collections allow one to use a type of collection to store the multiple versions, such as arrays, lists, and maps. This dynamic approach allows runtime addition of new versions, where the overhead of adding is related to the “insert” method of the collection. It is easier to turn off adaptation and it allows more flexibility in terms of adding, reassigning and removing algorithms, with the cost of possibly being slower than the conditional branching approach. Adaptation Decision is also limited to the key type of the collection, e.g. indices for lists and object keys for maps. This limitation can be solved in some cases with mathematical expressions.

Figure 4.7 shows an example of a strategy that generates a map at compile-time (line 9) with the algorithms to use and the scenarios in which they execute (lines 10 to 15) and a fallback method in case the scenario does not apply to any algorithm (line 16). The map is then introduced in the main code (line 18) and at runtime, the program just needs to access that map (line 23). This approach is ideal when one needs to update the algorithms to use and even update the scenarios in which the existing algorithms are selected.

```java
select method as targetMethod with name == targetName
introduce interface in app (name= "FuncInterface")
introduce method in interface (signature= targetMethod.signature)

select call with name == targetName //all calls in the app
extract call to variable named func of type interface
replace call with func.invoke(args= call.args)

algorithms = new Map(); //create a map
select method with name = alg1
algorithms.put(scenario1, method); //map scenario 1 to algorithm alg1
select method with name = alg2
algorithms.put(scenario2, method);
select method with name = alg3
algorithms.put(scenario3, method);
algorithms.orElse(targetMethod); //use original method as fallback

introduce algorithms in app as map
select func //the adaptation point
select updatePoint //user-defined update point
select relevantInfo //necessary runtime information
execute before updatePoint
func = algorithms.get(relevantInfo);
end execute
```

Figure 4.7. A strategy similar to the strategy in Figure 4.6 that uses a map instead of a static switch, mapping a predefined scenario to the corresponding algorithm.

Algorithm exploration is also possible and it is similar to the knobs approach, defining strategies to first explore the available algorithms and then map them to the desired scenario. The algorithms are stored in a list and every time a new scenario appears a new exploration is launched. During this exploration phase, each of the algorithms is sampled a number of times in order to obtain their score for that condition. The algorithm with the best score is stored in the map using the scenario data as its key. When the scenario exists in the map then the mapped algorithm is used.
Now the main problem is how to deal with executions with alternative scenarios, i.e., what happens if a different scenario executes while exploring another scenario? Then multiple explorations are needed, one per scenario. As the code is becoming more complex, it is ideal to start moving some programming logic to other modular units. This means moving the fields and adaptation code to a different class and at runtime, a new instance is produced per new scenario. The adaptation code in the main program changes to a check over the map to see if it contains an algorithm to the current scenario and if true that algorithm is returned, otherwise, the adaptation fetches an exploration for that scenario, initializing a new one if the scenario does not yet exist.

Figure 4.8 and Figure 4.9 show the definition of this strategy. Figure 4.8 starts by extracting the functional interface and the method call to a variable (lines 2 and 3). Then, it generates a new class, named Exploration, that will contain all exploration-related components and a list of algorithms that can be explored (lines 5 to 28). The search scheme, i.e. the way to explore and to change algorithms, is defined as the getAlgorithm method of this class (lines 12 to 27). The search scheme used in this strategy was to try all the available algorithms (line 22) and then stop and return the best algorithm (lines 18 to 21).

```java
select call with name == targetName
extractFunctionalInterface from call as FuncInterface
extract call to variable named func of type interface

introduce class in app (name= "Exploration") as ExplorationClass
introduce field in ExplorationClass (name="explore", type=boolean, init=true)
introduce field in ExplorationClass (name="algorithms", type=List<FuncInterface>)
introduce field in ExplorationClass (name="bestAlg", type=FuncInterface)
introduce field in ExplorationClass (name="bestScore", type=Score)
...
introduce method in ExplorationClass
public static FuncInterface getAlgorithm(){
    if(numSamples > LIMIT) {
        if(iScore < bestScore) {
            //criteria
            bestAlg = currentAlg;
            bestScore = currentScore;
        }
        if(!algorithms.hasNext()) {
            explore = false;
            return bestAlg;
        }
        currentAlg = algorithms.next();
        currentScore = new Score();
        numSamples = 0;
    }
    return currentAlg;
}
end introduce
```

Figure 4.8. Compile-time generation of a new interface and a class containing the exploration control fields and a method performing the search scheme.

Figure 4.9 continues the strategy specification adding a map to control the known scenarios and corresponding algorithms (line 2), a map to store explorations for unknown scenarios (line 3) and a timer to measure the target update point (line 4). The adaptation code now is as simple as verifying if the algorithms map has the current scenario and return the algorithm (lines 12 and 13). Otherwise, it uses an exploration scheme to find the proper algorithm for that scenario (lines 14 to 19). The measuring point is then measured (lines 22 to 24) and if the scenario is under exploration then add this new measurement to the algorithm score (lines 25 and 26).
Designing and Mapping Runtime Adaptivity Strategies

```
select call with name == targetName in class
introduce field in class (name="algorithms", type=Map<Scenario,FuncInterface>)
introduce field in class (name="explorations", type=Map<Scenario,Exploration>)
introduce field in class (name="timer", type=Timer, init="new Timer()")

select func //the adaptation point
select updatePoint //user-defined update point
select measurePoint //user-defined measure point
select scenario //necessary runtime information
execute before updatePoint
  Exploration exp;
  if(algorithms.contains(scenario))
    func = algorithms.get(scenario);
  else {
    exp = explorations.get(scenario);
    func = exp.getAlgorithm();
    if(exp.explore == false)
      algorithms.put(scenario,func));
  }
end execute
execute around measurePoint
  timer.start();
  proceed(); //proceed with normal execution of the measure point
  timer.stop();
  if(exp != null)
    exp.currentScore.add(timer.getTime());
end execute
```

Figure 4.9. Strategy for runtime exploration of algorithms based on scenarios only known at runtime. This strategy uses the functional interface and exploration class from the strategy shown in Figure 4.8.

The discussed algorithms are considered simple exploration algorithms. There are other algorithms which usage allows more adaptation opportunities, namely tunable algorithms, adaptive algorithms and generative algorithms. Table 4-I shows a description of the four types of algorithms available in the proposed approach. Tunable algorithms have one or more associated knobs which values influence the performance of that algorithm. In a simple way, they are algorithms to which no modifications are made and the actual adaptivity is fully focused on the target knobs which values influence the execution of that method. The tiled loop of Figure 4.2 is an example of an algorithm that has the block size as the associated knob. Adaptive algorithms are algorithms that will have their source code redefined at runtime, based on the strategy criteria. Generative algorithms are similar to adaptive algorithms with the difference that instead of redefining the target algorithm, new algorithms are generated at runtime, also based on the strategy criteria. These adaptive and generative algorithms are further described in the following section.
Table 4-I. Description of the types of algorithms one may use in an algorithm selection strategy.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>Algorithms not requiring anything from the strategy at runtime to be executed</td>
<td>Original version of an algorithm</td>
</tr>
<tr>
<td>Tunable</td>
<td>Algorithms with knob(s) that at runtime can be tuned to improve its execution</td>
<td>A knob that determines the number of iterations of a loop</td>
</tr>
<tr>
<td>Adaptive</td>
<td>Algorithms which source code is modified at runtime</td>
<td>Unroll a loop of the algorithm based on a runtime criteria</td>
</tr>
<tr>
<td>Generative</td>
<td>Algorithms that are generated at runtime</td>
<td>Maintain multiple versions of an algorithm if different scenarios require different versions</td>
</tr>
</tbody>
</table>

This type of algorithms makes the design of an exploratory strategy even more complex as mixing the various types of algorithms requires a mix of algorithm exploration and knob(s) exploration. Meaning that the exploration method not only has to consider the list of algorithms to use but also the specific exploration an algorithm may have. Figure 4.10 shows an example of a strategy that extends the algorithm strategy of Figure 4.8 and Figure 4.9 to use algorithms that contain linked knobs. The strategy in this example uses an algorithm that requires the tuning of a knob to improve its performance. This means that instead of just sampling this algorithm once, it will be sampled for each value the knob can take. Besides the list of algorithms, the strategy adds a map to the exploration class that maps a given algorithm to the corresponding knob exploration (lines 1 and 4). When a new version of an algorithm is requested, i.e. the current number of samples has been reached and the `getAlgorithm` method (lines 7 to 32) is invoked, before getting a new algorithm (line 23 to 27), the exploration verifies if the current algorithm has a knob (line 13). If so, it requests for a new value to the knob, applies that new value over the knob and uses the same algorithm (lines 16 to 20). When the knob exploration does not have more values to explore the next algorithm in line is returned.
introduce class in app (name= "KnobExpl") as KnobExplClass
introduce class in app (name= "Exploration") as ExpClass
introduce field in ExpClass (name="algorithms", type=List<FuncInterface>)
introduce field in ExpClass (name="knobs", type=Map<FuncInterface, KnobExpl>)
...
introduce method in ExpClass
    public static FuncInterface getAlgorithm(){
        if(numSamples > LIMIT) {
            if(iScore < bestScore) { //criteria
                bestAlg = currentAlg;
                bestScore = currentScore;
            }
            if(knobs.contains(currentAlg)) { //if algorithm has a knob
                knob = knobs.get(currentAlg);
                if(knob.hasNext()){
                    value = knob.getNextValue(); //get next value from the scheme
                    value.apply(); //and apply the value to the knob
                    currentScore = new Score();
                    numSamples = 0;
                    return currentAlg; //it is still the same algorithm
                }
            } else get next algorithm
        }
        if(!algorithms.hasNext()) {
            explore = false;
            return bestAlg;
        }
        currentAlg = algorithms.next();
        currentScore = new Score();
        numSamples = 0;
        return currentAlg;
    }
end introduce

Figure 4.10. Algorithm exploration with a map that associates a knobs exploration to an algorithm.

Unlike simple algorithms that do not require extra logic for its execution and algorithms with knobs that only require a configuration to assign parameters with specific values, the adaptive and generative algorithms require extra runtime execution logic. This logic includes mechanisms to generate the specialized versions of the code at runtime and to replace existing algorithms or instantiate new ones, for adaptive and generative algorithms respectively. The following section describes this code generation process by giving an overview over runtime code generation and specialization and by stating how this process should be specified in a runtime adaptivity strategy.

4.3 Runtime Generation and Specialization

Software applications are usually developed in a generic fashion, in a way that the same code can deal with different input data, execution, and system parameters. In some cases, however, specializing the program to predefined values can provide better program execution than the developed generic version, as exemplified by Renaud Marlet (Marlet, 2013):

“...If we have a program, on one hand, and a context for its execution, on the other (...) we seek to create a new program, whose behavior is identical to that of the original program (...) but which performs better because it is specialized to that particular execution context.” (Marlet, 2013)
Designing and Mapping Runtime Adaptivity Strategies

An application can take advantage of the information available at runtime such as input data, program and system configurations, to improve its efficiency. Certain type and range of input values can allow the execution of specialized versions of a program, where its performance is considerably better than the original (generic) version. Hence, an application can take advantage of possible specialized software versions, each version optimized for a dataset.

Consider a hypothetical method $M$, which one intends to specialize according to runtime information. A first naive approach for specialization would be the implementation of several versions ($V1$, $V2$, ..., $Vn$), each with a set of applied optimizations, and targeting a different set of range values ($V1$ for $D1$, $V2$ for $D2$, ..., $Vn$ for $Dn$). Following, one replaces all the invocations to this method with a switch containing $N$ cases, each branching to a specialized version (case($D1$) $V1$; case($D2$) $V2$ case...). For each $N$ specialized versions of a method, $N + 1$ cases would be required. Figure 4.11(a) demonstrates this approach.

The main issue of having all $N$ versions precompiled is when $N$ is too large or unknown (e.g. only a range is known). Some algorithms use a particular set of operations that do not allow a complete specialization to address all possible combinations, allowing only partial specialization, as in some image processing (Pavlidis, 2012). Some image processing algorithms use sliding windows that contain a set of coefficients to process the target image, and providing multiple code versions for each possible coefficients may imply an oversized program and even possible code size explosion.

The proposed approach intends to avoid the generation of all of these specialized versions as much as possible. Instead, the required version should be defined during the application execution (i.e., using the approach in Figure 4.11(b)). Considering that specific program data (such as arguments) usually define the branches and iterations executed, the dynamic generation of a specialized version, according to that program data, should provide better performance than a generic version and reduce space overhead for multiple versions. However, runtime generation
Designing and Mapping Runtime Adaptness Strategies

can become itself an overhead if the application requires to halt for a new version. Hence, this generation overhead is another concerning issue.

Code specialization is subjective to the program code/structure and the contextual information. How a program is specialized and which parts of the program can be specialized, require certain program and contextual knowledge.

Compile-time specialization has its advantages, as it provides more time to perform code generation and does not influence the code execution. However, the only information available at compile-time is from profiling and/or from using speculative information. The proposed approach allows specialization at runtime, by generating the requested version with the given runtime information. It is important to mention that dynamic code generation requires more planning than an offline generation, especially because the online generation may significantly affect program performance.

The proposed approach aims to provide user-control of program specialization, where one can define specialization on the DSL and explore compiler optimizations that can take advantage of information that was not accessible at compile-time. Table 4-II describes a number of compiler optimizations relevant to this concept, including examples of how specialization may profit from code optimization guided by contextual information.

These optimizations alone may not suffice to ideally improve the performance of a program, which is why combining these different optimizations may provide better strategies for program improvement. Furthermore, the impact of an optimization may differ on the target system. For instance, loop interchange may have different impact when a specific kernel executes in a CPU or in a GPU. Therefore, a strategy for code specialization must be defined taking into account several factors, including target application, target system, and contextual information.
**Table 4-II. Example of code optimizations used for program specialization.**

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Constant Propagation</strong></td>
<td>Replace the use of variables, fields and array accesses with the corresponding value if the value is constant</td>
<td>When the value remains unaltered for the specialized version</td>
</tr>
<tr>
<td><strong>Loop Unroll</strong></td>
<td>Unroll a loop with a factor of $K$</td>
<td>Fully unroll of a loop, or with a factor of $K$ (for a large number of iterations)</td>
</tr>
<tr>
<td><strong>Loop Tiling</strong></td>
<td>Partition a loop in small blocks of size $B$, ensuring that data stays in the cache for subsequent accesses</td>
<td>Size of block $B$ can be defined using the number of loop iterations and the size of the cache</td>
</tr>
<tr>
<td><strong>Loop Interchange</strong></td>
<td>Change the order of two loops in nested loops</td>
<td>Shape of data and pattern accesses determine loop interchanges</td>
</tr>
<tr>
<td><strong>Data reuse</strong></td>
<td>Keep the last $N$ values accessed stored in local variables</td>
<td>If each iteration accesses $N$ previous memory locations, store the values in the current iteration access for the subsequent iteration</td>
</tr>
<tr>
<td><strong>Scalar Replacement</strong></td>
<td>Replace memory accesses with a local, scalar, variable, and access only once that memory location</td>
<td>Loops with continuous store in an array position, replace with a scalar variable and store only once (outside the loop)</td>
</tr>
<tr>
<td><strong>Expression Simplification</strong></td>
<td>Simplify expressions by replacing with an equivalent, more efficient, expression</td>
<td>Remove unnecessary operations using a zero, or even an entire expression</td>
</tr>
<tr>
<td><strong>Dead code Elimination</strong></td>
<td>Remove code that will never be executed, i.e., the program flow does not cover that code</td>
<td>Remove conditional branches that are never executed after specialization</td>
</tr>
<tr>
<td><strong>Function Inlining</strong></td>
<td>Replace a function call with the body of the called function</td>
<td>Avoid calls and returns and may provide more potential for specialization</td>
</tr>
<tr>
<td><strong>Function Cloning</strong></td>
<td>Generate a copy of a function</td>
<td>Useful to maintain a function with different optimizations in each clone</td>
</tr>
<tr>
<td><strong>Multiple Versions</strong></td>
<td>Manage a set of algorithms and specialized clones</td>
<td>Manage a set of versions of an algorithm, each with a loop unrolled with a specific factor</td>
</tr>
<tr>
<td><strong>Algorithm Selection</strong></td>
<td>Select the best performing algorithm for a current scenario</td>
<td>The sorting process is a common example in which the existing algorithms perform differently, for instance, with the number of elements to sort</td>
</tr>
</tbody>
</table>

By using a lower-level approach (such as Bytecodes (Venner, 1996)), despite making harder code generation, it makes the generation faster as there is no need to perform compilation from source code to bytecodes. Hence, in order to reduce overhead and ease the code generation, the proposed approach uses code specialization based on bytecode templates. The templates are expected to be portions of bytecode instructions with specific “operations” intertwined that allow code generation and the application of compiler optimizations. These operations can be seen as pragmas requiring some input values to apply the specified optimizations. Figure 4.12 illustrates how the templates might be defined. In Figure A.1 the source code of smooth algorithm was shown and Figure A.2 shows a specialized version of that code based on applying loop unrolling and constant propagation transformations. In Figure 4.12(a) one can see a parcel of the original Java bytecodes (Lindholm et al., 2014) related to the innermost loops of the smooth algorithm. The bytecodes in lines 6, 10 and 13 to 17 access the iterators of these innermost loops and the coefficients window. Based on the original bytecode instructions, the bytecode template in Figure 4.12(b) fully unrolls the two loops and replaces the coefficients window access with the corresponding coefficient, accepting the coefficients window as the input matrix. The two #for tags in lines 1 and 2 generate the code repetition depicted in lines 3 to 16 according to the window size, while the use of the iterating variables and coefficients window access is replaced with the corresponding value.
Figure 4.12. Java bytecode instructions snippet of the smooth method related to the two innermost loops and the inner operation: (a) original bytecodes; (b) bytecode template.

Note that the contribution of this work is the use of a template-based approach independently on how the templates are designed. The design of the templates is an orthogonal problem and therefore the presented templates are simple prototypes to illustrate the proposed approach. The templates have a simple design and small extensions so that a developer with knowledge on Bytecode instructions can easily understand these extensions. To ease the class modifications that occur at runtime, this approach requires an API with access to JVM functionalities.

When the templates are used in a strategy they are weaved in the target application together with the adaptation code. The way the templates are used depends if the strategy is dealing with adaptive algorithms or generative algorithms.

Adaptive algorithms focus on adapting the code of a target method. At runtime, the template-based approach generates new Bytecodes specialized to some inputs and replaces the current Bytecodes of that method. A model of a strategy for method adaptivity is shown in Figure 4.13. The strategy makes use of an API (line 1) that provides tools that dynamically generate Bytecodes based on the target method, a template and input values. The input values are directly related to the inputs of the template. The template and the target method are used to generate an adapter (line 4) that specializes the method based on runtime data (line 9). All the necessary engineering for code generation is masked with the use of the API, reducing the effort of the developer to completely define the generative code and the adaptation of the target method.
Designing and Mapping Runtime Adaptivity Strategies

```java
import API MethodAdapter
select template with name == myTemplate
select method with name == targetName
adapter = MethodAdapter.build(method, template)  //generates necessary
//adaptation code
select updatePoint  //user-defined update point
select scenario  //necessary runtime information
execute before updatePoint
    adapter.adapt(scenario)  //specialized version to given scenario
end execute
```

Figure 4.13. A strategy that specializes at runtime a target method based on a Bytecode template and some runtime data.

It is also possible to define a strategy that uses a search scheme and a template to explore different versions of a method. An example is the use of a template that addresses the loop unroll transformation and the unroll factor as inputs. At runtime one may feed this template with different factors to try and find the most suitable factor for a given scenario.

Adaptive algorithms are ideal when trying to specialize code to a specific scenario and at the same time to avoid having too many multiple versions. The problem is when the program has to deal with multiple scenario and so multiple versions of the method might be necessary. The generative algorithms solve this problem as they generate new anonymous instances of that algorithm instead of always rewriting the target code. The approach is very similar to the adaptive algorithm with the slight difference that instead of a method reference it needs a functional interface and the reference to the dynamically generated algorithm must be stored.

Figure 4.14 shows the design of a strategy that builds a method generator based on a template and a functional interface (line 4). A map is introduced in the target code to store for each scenario the generated version (line 8). When the map does not contain a version of a given scenario, the generator is used to generate a specialized version based on that input scenario (line 13 to 14). Maintaining a map avoids having to regenerate versions since the runtime generation has some impact on the performance.

```java
import API MethodGenerator
select template with name == myTemplate
select interface with name == targetName as FuncInterface
generator = MethodGenerator.build(FuncInterface, template)
select updatePoint in class  //user-defined update point
select scenario  //necessary runtime information
introduce field in class (name="algorithms", type=Map<Scenario,FuncInterface>)
execute before updatePoint
if(algorithms.contains(scenario)) {
    func = algorithms.get(scenario);
} else {
    func = generator.generate(scenario)  //specialized version
    algorithms.put(scenario,func);
}
end execute
```

Figure 4.14. A strategy that generates at runtime specialized versions of a method based on a Bytecode template and some runtime data.

Making decisions at runtime for code generation is a difficult task when dealing with unpredicted scenarios or environments. One way to decide which version to generate is to use an exploration approach in which a search scheme is used to provide the adaptivity with new
configurations to explore and sample and, based on that sampling, decide which configuration, and consequently, the generated algorithm is ideal for a given scenario. The concept of search schemes is discussed in the following section.

4.4 Search Schemes

A runtime adaptivity strategy that explores different configurations for a given scenario requires a search scheme. Search schemes determine how the values the target knob(s) can have are explored and provide configurations to apply to the knob(s). The most basic search scheme is a full, exhaustive exploration of all possible values within the knob(s) range(s) which is ideal in circumstances in which the exploration space is small. However, this approach decays and might be impracticable when dealing with big ranges or with multiple knobs.

Consider the example of Figure 4.4 in which a full exploration is made to find the best block size from a range of [8, 2048]. Since the exploration scheme is using the power of 2 values it takes 9 configurations to complete the exploration, but it is guaranteed that the best configuration is found. This number of configurations is not necessarily high and it may be acceptable in some circumstances. However, consider now that instead of having a single knob the adaptation deals with two knobs, each with the same range of [8, 2048]. This means 81 configurations have to be sampled before deciding which one is the best configuration. Increasing to three knobs means 729 configurations to sample. In normal execution, this is almost impracticable to perform.

Figure 4.15 shows an example of a full exploration using two knobs, J Block and K Block, both with the same range. The size of the bubbles depicts the speedup when running the application with the configuration defined by (J Block, K Block). The exhaustive approach indeed achieves the best approach but had to explore a total of 81 configurations and pass through 36 configurations with worse performance than a code version not considering the two knobs.

Figure 4.15 Execution time speedup obtained from an exhaustive exploration of all possible values for two block sizes of two tiled loops on a matrix multiplication algorithm.
As the exhaustive exploration might require too many configurations to explore, different types of search schemes that may fit better the developer needs and improve the exploration space can be used. Hence, it is important to provide the developers with different types of search schemes and allow them to specify new schemes or extend the existing ones. For example, by using the power of 2 values instead of 1-unit increment the developer significantly reduces the number of exploration points. These configurations can be used in an exploratory strategy and they are selected by the developer. Some examples of searching configurations are:

- **exhaustive**: all values are used;
- **specific increments**: define an increment reduces/increases the exploration points;
- **random values**: randomize configuration values based on a list or a knob range;
- **profile-based**: static definition of a set of values to evaluate based on offline profiling;
- **steepest descent**: search neighboring values and “jump” to a better neighbor;
- **local minimum**: fix parameter1, explore parameter2 and then explore parameter1 with best parameter2.

The use of a non-exhaustive exploration comes with the downside of possibly not finding the best configuration for the target goal. Figure 4.16 shows how the block sizes were explored for two tiled loops when executing the steepest descent (a) and the local minimum (b) search schemes. The steepest descent allows moving away from bad configurations since it starts with a decent configuration, i.e. configuration with good/acceptable performance, and to achieve a very good configuration with only 24 configurations sampled. This shows that the local minimum approach is advantageous to reduce the space exploration and that the starting value is very important in order to find adequate configurations. For instance, by starting with (8,8) would prevent the use of loop tiling since all values around these points would give worse performances. The local minimum search first fixes the K block with 128 and finds the best value of J Block and then searches for K block with J Block fixed with the value 2048. This scheme is able to achieve the best configuration with only 17 configurations and again due to the decent starting configuration.
Designing and Mapping Runtime Adaptivity Strategies

Figure 4.16 Execution time speedup obtained from configurations for two block sizes of two tiled loops on a matrix multiplication algorithm.

To show a perspective of how the steepest descent reduces significantly the exploration space Figure 4.17 depicts that same scheme applied to three knobs. In this case, the exploration space is reduced from 729 configurations to only 76 configurations, i.e., a reduction of almost 90%.
Although some of the configurations may not achieve the global best version, they may locate local minima which may be used as a reference of other explorations when activated. Hence, the starting point for a local minimum approach should be carefully selected. Some experiments in Chapter 7 consider this type of exploration scheme, using the steepest descent search in an algorithm with a tunable parameter.

### 4.5 Autotuning

Many approaches use offline profiling to determine the best scenarios (e.g., offline profiling-based autotuning (Gadioli et al., 2015)). However, this may be unfeasible when working with unpredictable environments and/or when targeting many/unknown platforms. These offline scenarios may serve as the baseline for pre-ordering of algorithms/versions and can be then extended with a runtime exploration to achieve ideal scenarios for a specific device. This exploration may be done by selecting, or generating, algorithms and/or specifying new values for knobs.

The previous sections already depicted some examples of how to define strategies that insert code to control an exploration. The strategy needs to be completely designed by the developer, which is ideal when he/she intends to have complete control over the adaptivity code. However, when dealing with more complex exploration strategies complicates the developer effort. Some details become difficult to program, such as how to explore a list of algorithms in which some may have associated knobs or may be generative or adaptive specialized algorithms. Also, the strategies become too big and more error-prone.

To smooth the developer experience, it is necessary an autotuner, a feature that allows the automatic tuning and/or exploration of knobs configurations and/or algorithm selection. Figure
4.18 shows a simplified version of the autotuning execution flow. The application requests a version from the Autotuner via the update method (1). The Version Manager keeps a channel with the generated versions and the best performing algorithm and is responsible to update the adaptation point with an exploratory/best version (2). When executing in exploration mode the Autotuner returns a version from the version manager, and when in the best mode it returns the stored version considered as the best version so far. The application feedbacks measurements to the Autotuner via its measurements channel (3). The measurements are analyzed (4) to decide if the explored version is better than the current best version (5). Finally, the Autotuner requests the Configurer for new versions (6), which are dispatched to the versions channel (7).

The autotuner has to be easy to build and easy to use in an adaptivity strategy. This approach takes full advantage of API functionalities to ease the effort of the developer, including APIs to build code generators, select search schemes, select measurers, among others. Code restructuring is still the first step in the adaptation strategy. This prepares the code so the autotuner can access information and adapt the application more easily. The development of autotuners should be very similar between strategies which facilitate the learning process and promotes strategy reusability. Furthermore, the autotuner should allow the "clone" of explorations to begin a new search for a different scenario, with a list of configurations ordered in a more ideal way, after learning from previous runs, reducing the exploration space.

When dealing with knobs, the autotuner focus on providing configurations for those knobs based on a search scheme. Figure 4.19 shows how an autotuning strategy for searching the best configuration for two knobs providing the lowest execution time. Lines 1 to 4 include the imports necessary for the exploration and line 5 represents hidden code restructuring that would be necessary for the adaptation. Then, the strategy selects the two parameters to be used as knobs and selects the data representing the runtime scenario, the update point and the measuring point. A range is specified in line 11 that defines the boundary values and the decrement and increment functions to decrease/increase the current configuration value. In lines 12 and 13, two knobs with that range are instantiated and lines 14 to 20 build the autotuner. First, the autotuner is built by specifying what is the adaptation target, which in this case is knob1 and knob2. Then data to which exploration is mapped has to be specified, i.e., how an exploration for a specific scenario is started. The search scheme and the goal also have to be specified and for that Scheme and Measurer APIs are used. In the example, the steepest descent was used as the search scheme and
the goal was defined as “minimize the average execution time”. The update and measurement point have to be specified to add the adaptation code in the application.

```java
import API Autotuner
import API Knob
import API Scheme
import API Measurer
...
select field (name == “param1”) as field1
select field (name == “param2”) as field2
select scenario //necessary runtime information
select updatePoint //user-defined update point
select measurePoint //user-defined measurement point
range = new Range(8, 2048, x=> x >> 1, x=> x << 1)
knob1 = new Knob(field1, range) //specify two knobs with a [8,2048] range
knob2 = new Knob(field2, range) //each targeting a field
Autotuner
.adapt knob1, knob2 //adapt the target knobs
.foreach scenario //one exploration per scenario
.with Scheme.SteepestDescent //explore the knobs with a given scheme
.goal Measurer.AverageTime.Minimize //time as the score and fastest as the goal
.update before updatePoint //change the knobs values in this point
.measure around measurePoint //measure the score in this point
```

Figure 4.19. A strategy building an autotuner that searches for each possible runtime scenario the best configuration of two target knobs that improves execution time.

The autotuner API generates a class that contains the autotuning code, including all the necessary exploration logic and management, and injects the adaptation code in the main program. Figure 4.20 shows a snippet of the code weaved on the update and measurement points and, compared to Figure 4.4 and Figure 4.5, the injected code is really small, as all the adaptation logic is defined inside the Autotuner class.

```java
Autotuner autotuner = new Autotuner();
...
autotuner.apply(scenario); // apply configuration based on current scenario
//updatePoint
...
autotuner.startMeasure();
//measurePoint
autotuner.stopMeasure();
```

Figure 4.20. A snippet of the injected autotuner code to control the knobs configuration and obtain runtime feedback with the specified measurement.

Building an autotuner for exploring algorithms is more complex than the autotuner of knobs. In this case the developer has three types of algorithms to use: simple algorithms, algorithms with knobs and generative/adaptive algorithms. Simple algorithms can be added to the autotuner with nothing more than a reference to that algorithm. Algorithms with knobs are added with their reference and the references to the target knobs and the search scheme to use for those knobs, or the value reference to use if no exploration is necessary for that knob. An adaptive/generative algorithm requires that its generator has to be passed as the reference and, similar to the algorithm with a knob, a search scheme, or a value reference, has to be specified.

The usual specification of an autotuner for algorithms exploration is depicted in Figure 4.21. Compared to the knobs autotuner in Figure 4.19, this strategy now includes two more APIs, Algorithm and MethodGenerator. The autotuner for algorithm requires that a functional
Designing and Mapping Runtime Adaptivity Strategies

interface has to be generated based on the target function call (line 13) and that call to be extracted to a field (line 14), which will be the adaptation point for the autotuner (line 19). A knob is instantiated to be used by an algorithm (line 16) and also an instance of a generator added (line 17). The generator uses the extracted functional interface and template selected by the developer. The autotuner is built in lines 18 to 28 and, compared to the autotuner in Figure 4.19, the main difference is the set of algorithms placed instead of the search scheme in the “with” statement. The first two algorithms, alg1 and alg2, represent simple algorithms which do not require any special treatments to be executed. Then, the third algorithm, alg3, requires the use of a knob, knob1, and specifies that this knob must be explored with the steepest descent scheme. The last algorithm is, in fact, a generative algorithm that generates new algorithms using the data of the current scenario.

```java
import API Autotuner
import API Knob
import API Scheme
import API Measurer
import API Algorithm
import API MethodGenerator
...
select scenario //necessary runtime information
select updatePoint //user-defined update point
select measurePoint //user-defined measure point
select field (name == "param1") as field1
select call with name == targetName
extractFunctionalInterface from call as FuncInterface
extract call to variable named func of type interface
range = new Range (8, 2048, x -> x >> 1, x => x << 1)
knob1 = new Knob(field1, range)
generator = MethodGenerator.build(FuncInterface, template)
Autotuner.
.adapt func
.foreach scenario
.with
.algorithm alg1,
.algorithm alg2,
algorithm alg3 using knob1 with Scheme.SteepestDescent,
algorithm generator using scenario
.goal Measurer.AverageTime.Minimize
.update before updatePoint
.measure around measurePoint
```

Figure 4.21. A strategy building an autotuner that explores four different algorithms: two simple algorithms, an algorithm with a knob and a generative algorithm.

The autotuning is a high-level abstraction for automatic runtime adaptivity that uses the previously described adaptivity approaches. The specification of an autotuner in a runtime adaptivity strategy is based on the build pattern (Gamma, 1995) and, at weaving time, generates all the code for runtime autotuning, including the exploration management, and inserts in the target application. This approach simplifies the specification of strategies and bypasses the requirement of the strategy developer to specify the actual runtime adaptivity decision code, as it is the responsibility of the autotuner “builder” to generate that code.
4.6 Summary

This chapter presented the approach for the design of runtime adaptivity strategies and it is divided in the different levels of adaptivity that the approach delivers. For each adaptivity level, the compile-time restructure and the description of runtime adaptivity was carefully discussed and exemplified by strategies defined in pseudo-code for easier understanding.

It is shown how to design adaptivity strategies at the software and optimization knobs, including strategies that explore different configurations for the knobs and try to find the best configuration based on a target goal. Then, how to make runtime algorithm selection was discussed and how to make runtime exploration of algorithms for a specific scenario. After that, a template-based runtime code generation approach was presented. Here it was explained that the template-based approach has to be based on generating low-level code, namely Bytecodes, as it is more efficient at runtime. The design of runtime adaptivity strategies for runtime generation was also discussed.

In the end, an autotuning approach was presented to deliver a more efficient way of designing strategies for the exploration of configurations for different target scenarios.
Designing and Mapping Runtime Adaptivity Strategies
Programming Runtime Adaptivity in Java with LARA

Contents

5.1 The Java API ......................................................................................................................... 104
5.2 Kadabra Weaver and API .................................................................................................... 107
  5.2.1 Language Specification ................................................................................................. 107
  5.2.2 Aspects API ..................................................................................................................... 109
5.3 Strategies Specification ....................................................................................................... 110
  5.3.1 Runtime Adaptivity with Knobs ..................................................................................... 111
  5.3.2 Runtime Algorithm Selection ....................................................................................... 112
  5.3.3 Runtime Generation ....................................................................................................... 113
  5.3.4 Autotuning ..................................................................................................................... 114
5.4 Summary ............................................................................................................................ 116
This chapter presents the developed weaver engine and features of the Java API. It shows how LARA can be used as the DSL to program runtime adaptivity strategies using a Java-to-Java weaver engine and a runtime adaptivity API. The features of the runtime adaptivity API are discussed, followed by how the Kadabra weaver and the Kadabra API can be used to control the runtime adaptivity API by means of LARA aspects.

5.1 The Java API

To help developers with respect to runtime information retrieval and autotuning algorithms, it is proposed the use of a Java API. Developers can extend this API, select another API or develop their own search algorithms. The API can be fully controlled by LARA aspects and provides useful features for runtime information fetch and adaptivity. Some of the main features are the following:

- access system information: number of CPUs, number of threads or cache size;
- runtime monitors: measure time, energy and value range;
- concurrency utilities: concurrent channel for communication between threads;
- configuration providers: implementations of search schemes;
- knob ranges: controls the ranges of a knob and its increment/decrement step;
- template-based code generator: method bytecode replacement and/or inclusion of new methods;
- autotuning manager: explore and sample algorithms and/or software knobs using time/energy monitors.

A runtime adaptivity strategy bases its work in the information that can only access at runtime. This information can be data obtained from the program itself or data about the execution environment. When dealing with different target environments, access to the execution environment data becomes a fundamental issue. With this API it is possible to access information about the CPU (including the number of CPUs, the number of threads and the size of the caches), the memory usage, available energy and execution time.

The runtime monitors allow monitoring the performance of the application at runtime. These monitors can be used to make runtime decisions such as apply a specific adaptation when the program is executing more than a given time, or turn off an exploration when the battery is too low. The monitors are also used as measurers for exploration strategies, e.g. to try to minimize execution time or energy consumption.

The concurrency utilities serve as middleware for the developer to add concurrency in the adaptation. These utilities include communication channels that automatically deal with concurrency issues. Concurrent communication channels are important when trying to develop strategies which adaptation logic should execute parallel to the main program logic (as shown in Figure 3.17), or simply if the user wants to parallelize some data being channeled in the program. Two examples of effectively using the concurrent channel are having one channel to communicate runtime measurements and a channel used to communicate algorithms that are generated concurrently to the program execution.
Configuration providers are implementations of some of the discussed search schemes and are used in exploration strategies. Currently, the available search schemes are full exploration, steepest descent, local minimum, random and user-defined. A search scheme accepts as input a list of values or a knob range. A user-defined scheme allows one to completely control how configurations are deployed and can even combine search schemes when dealing with multiple knobs. For instance, one may define its own scheme that uses the steepest descent scheme for a knob and the local minimum for another knob.

The knob range is used to control the values a knob can have. The range is defined with the lower limit and upper limit and the starting value and ascending/descending steps can be defined by the developer.

The templates are fed to the approach are converted to Java code using the ASM framework that, by means of a Java bytecode API, is able to dynamically generate new versions.

For the template-based code generation approach, the API has three main classes: AsmUtils, MethodAdapter and MethodGenerator. The approach uses template-based Bytecode generation that takes advantage of Java mechanisms for dynamic class loading and instrumentation. The ASM framework (Bruneton et al., 2002) is used as middleware for the definition of the Bytecode templates dynamically invoked during execution. ASM is a reliable tool to deal with dynamic adaptation, showing the best performance when compared to approaches such as BCEL (Dahm, 1999) and SERP (White, 2002). The tool allows the visit of any bytecode instruction with a simple and high-level approach, allowing, at the same time, the insertion of low-level (bytecode) instructions. Since the main concern is focused on optimizing running code, Bytecode manipulation is limited to a method’s body.

Two ways of method generation are available: MethodAdapter and MethodGenerator. Both of these generators are built with a template, developed using the ASM framework, and at runtime, they generate code based on input data. A code generator is represented as a java.lang.instrument.Instrumentation class, which is responsible to adapt the target class with the new implementation generated with the template and runtime information. This instrumentation class is added to the JVM’s Instrumentation manager. When a class load request occurs, the ClassLoader (Chan et al., 2002) will invoke the instrumenting class, and a new version replaces the original version.

The difference in the method generation types is that MethodAdapter directly changes a target method of an existing class, while MethodGenerator creates new anonymous classes with the generated methods. The AsmUtils is a class providing utility mechanisms to add the method generators in the execution and it is based on the concept of Java agents (Binder et al., 2007). This class is also used to request the adaptation of a target class.

The autotuner API is a little more complex and requires additional classes and components. First of all, the approach considers two different types of autotuner: autocompiler with knobs called KnobExploration and autotuner for algorithms selection named AlgorithmExploration.

A KnobExploration allows exploration of a knob or multiple knobs, with a given search scheme, for different target scenarios. The KnobExplorationSupervisor is an abstract class responsible to overlook the knob explorations for each scenario. The supervisor uses three Generic types: D, C, and M. D represent the type of data of the studied scenarios, C is the type of
the configurations for the target knob(s) and $M$ is the type of the measurements used (aka the type of the score of a configuration).

Every time a new scenario of type $D$ occurs, a new KnobExplorationManager is used for that scenario $D$. This manager controls the exploration by providing new configurations to explore and keeping the score for each sampled configuration. It requires a search scheme, provided by the supervisor, to make new configurations to sample. An exploration manager can be run in four different modes: best, sampling, resampling and halt. The best mode uses the configuration which has the best score so far. In sampling mode, the exploration manager keeps providing new configurations to sample until no more configurations can be built or the manager mode is changed. Resampling mode allows the exploration to rerun the exploration using previous knowledge from the already sampled configurations. This mode is important when dealing with very dynamic environments for which the best configuration may vary during the same execution. In halt mode, the exploration is ignored and a default configuration is always returned.

KnobExplorationSupervisor is the only class one has to extend to build an autotuner for knobs. The class requires the implementation of the following methods: searchSchemeProvider, defaultConfig and getScoreEvaluator. The first method is a method to provide a search scheme every time a new KnobExplorationManager starts. The second one returns a default configuration to be used as a fallback or when the manager is in halt mode. The last method specifies which type of scorer evaluator the manager uses to compare configurations. This evaluator keeps track of measurements scores and has a goal by which the scores are compared. For instance, if the objective is to minimize execution time then the score evaluator will compare scores and assign the smallest score as having the best configuration.

During the execution, to get a configuration from the supervisor one must use the getConfiguration($D$) method that expects as input the current scenario data and returns the current configuration being sampled. The application applies the current configuration and must feedback the ScoreEvaluator of that exploration with the score the configuration obtained during execution. For instance, if the goal is to measure execution time then the evaluator would receive that time as feedback by invoking configuration.addMeasurement($M$).

An AlgorithmExploration is very similar to the KnobExploration however it has a few extra details. It still uses a supervisor (AlgorithmExplorationSupervisor) and an exploration manager (AlgorithmExplorationManager) and a configuration is now an interface of Algorithm and can be one of the following: a simple algorithm, an algorithm with a knob, an adaptive algorithm, a generative algorithm or a user-defined algorithm. The supervisor uses three Generic types: $D$, $A$, and $M$. $D$ still represent the type of data of the studied scenarios, $A$ is the actual type of the target functional interface and $M$ is the type of the measurements used (aka the type of the score of a configuration).

Algorithm is an interface that contains two methods: get and apply. The get method returns the actual algorithm to execute (i.e., an instance of $A$) while the apply method depends on the subtype of Algorithm. For algorithms with knobs, it serves to apply the configuration in the target knob. The adaptive algorithm changes the Bytecodes of the target method before it is executed and the generative algorithm is used to generate a new method with a provided configuration that can then be accessed by the get method. The user-defined algorithm gives the developer the freedom to decide what to do in the apply method. This means that the apply method if it is needed, it should always be executed before getting the actual algorithm. These
two methods are separated to give the developer autonomy to decide when the apply is executed and when the algorithm should be actually used.

The following section describes how this API can be controlled by LARA strategies and added to the main program logic.

5.2 Kadabra Weaver and API

To apply a given LARA strategy to a target application a weaving engine is required. The proposed approach requires a LARA weaving engine targeting the Java language with a set of join points, attributes and actions, such as the insert action and actions related to code transformations (such as loop tiling and method cloning). Kadabra is a new source-to-source weaver that extends the Spoon framework (Pawlak et al., 2016) with LARA weaving capabilities and with code transformations. Kadabra is able to manipulate the input Java program using the intermediate representation provided by the Spoon framework and to control the weaving process with the LARA framework.

This static weaver is responsible for the application of compile-time changes, similar to the current existing LARA weavers, and to weave the application in order to allow runtime adaptation. The weaving sites are the responsibility of the weaver, opposite to the static weaving. This weaver is developed using the tools and features by the LARA framework, which contains a semi-direct integration with the LARA interpreter. This feature allows easier development and integration of different weaving environments for LARA, providing a bridge between the transforming tool and the interpretation of LARA aspects.

5.2.1 Language Specification

For manipulating the target code, it was selected Spoon (Pawlak et al., 2016) as the source-code transforming tool to integrate with the static weaving engine. Spoon is a tool for Java source-to-source transformations and analysis, with a complete and fine-grained meta-model to access any program elements, such as classes, methods, fields, statements and expressions.

The weaving engine includes the integration between the current framework existent in LARA and the Spoon Java-to-Java compiler framework (Cardoso, Carvalho, Coutinho, et al., 2012, Pawlak et al., 2016). This integration relies on the LARA weaver generator that uses the specification (join point, attributes and actions models) of the target language to generate a Java abstract hierarchy. This generated weaver interfaces between the LARA interpreter and the Java-to-Java tool (responsible for parsing and adapting the target Java code).

Table 5-I shows some of the join points implemented in the Kadabra weaver, where the join point app depicts the root of the application, i.e., the first element to use in a select statement. These join points have a direct relation with the nodes of the syntax tree built by the Spoon Library. It is possible to observe that one can access fine-grain locations in the code, including conditional expressions of if and loop statements and even variable usages (var) inside an arbitrary expression. The extends column means that join point extends all functionality of that parent join point. For instance, field extends declaration means that field can select the join point init and all the attributes and actions of declaration are also accessible in the join point field.
Table 5-I. List of some of the available join points and the join points that can be selected from those.

<table>
<thead>
<tr>
<th>Join point</th>
<th>Extends</th>
<th>Selects (join point of type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>app</td>
<td>file</td>
<td>type, class, interface, pragma, comment</td>
</tr>
<tr>
<td>file</td>
<td>type</td>
<td>field, method, pragma, function (method), comment</td>
</tr>
<tr>
<td>class</td>
<td>declaration</td>
<td>init (expression)</td>
</tr>
<tr>
<td>field</td>
<td>declaration</td>
<td>param (declaration), body</td>
</tr>
<tr>
<td>method</td>
<td>statement</td>
<td>statement, firstStmt (statement), loop, if, declaration, assignment, return,…</td>
</tr>
<tr>
<td>body</td>
<td>statement</td>
<td>init (statement), cond (expression), step (statement), expr (expression), body</td>
</tr>
<tr>
<td>loop</td>
<td>statement</td>
<td>cond (expression), then (body), else (body)</td>
</tr>
<tr>
<td>if</td>
<td>var, call</td>
<td>var, expression, rhs (expression)</td>
</tr>
<tr>
<td>statement</td>
<td>expression</td>
<td>expression, var, arrayAccess, binaryExpression</td>
</tr>
<tr>
<td>assignment</td>
<td>expression</td>
<td>target (expression), index (expression)</td>
</tr>
<tr>
<td>expression</td>
<td>expression</td>
<td>target (expression), arg(expression)</td>
</tr>
</tbody>
</table>

Table 5-II shows the join point information one can access in a LARA aspect. The attributes vary from direct information of the join point, such as a variable name or the type of loop, to information on the join point relative position to other join points, e.g. the rank of a loop/if statement or the nested level of a loop. The first line depicts attributes that are accessible in any of the join points, meaning that in any join point one can retrieve the source code or access its join point ancestor. The join point class is an example of a missing join point in Table 5-II since it does not have attributes of its own but inherits the attribute of join point type.

Table 5-II. List of accessible attributes of some of the join points.

<table>
<thead>
<tr>
<th>Join Point</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>srcCode(String), ancestor(joinpoint), type(String), line(int)</td>
</tr>
<tr>
<td>file</td>
<td>name(String), path(String), dir(String), package(String), numClasses(int), numInterfaces(int)</td>
</tr>
<tr>
<td>type</td>
<td>name(String), qualifiedName(String), superClass(String), package(String), interfaces(String[]), modifiers(String[]), javadoc(String), isSubtypeOf(boolean), type(String)</td>
</tr>
<tr>
<td>method</td>
<td>name(String), returnType(String), isStatic(boolean), declarator(string), privacy(string), toReference(string), toQualifiedReference(string)</td>
</tr>
<tr>
<td>field</td>
<td>declarator(string), staticAccess(string)</td>
</tr>
<tr>
<td>declaration</td>
<td>name(String), type(String), isArray(Boolean), isPrimitive(Boolean), completeType(String)</td>
</tr>
<tr>
<td>statement</td>
<td>kind(string)</td>
</tr>
<tr>
<td>loop</td>
<td>type(LoopType), rank(string), nestedLevel(int), isInnermost(boolean), isOutermost(boolean), controlVar(string)</td>
</tr>
<tr>
<td>if</td>
<td>rank(string)</td>
</tr>
<tr>
<td>var</td>
<td>name(string), reference(read</td>
</tr>
<tr>
<td>call</td>
<td>name(string), qualifiedDecl(string), declarator(string), executable(string), target(string), targetType(type), returnType(string)</td>
</tr>
<tr>
<td>assignment</td>
<td>operator(string)</td>
</tr>
</tbody>
</table>
Some of the actions available in the Kadabra weaver are depicted in Table 5-III. All of the join points have the insert action available and most of the individual actions were developed with the intention to help in the definition of runtime adaptivity. For instance, the extractFunctionalInterface action allows one to generate a new functional interface based on the target method, the extractToFile action allows any arbitrary expression to be extracted to a local variable and tile transformation exposes the block size as a variable or field so it can be changed at runtime.

<table>
<thead>
<tr>
<th>Join Point</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>insert</td>
</tr>
<tr>
<td>app</td>
<td>newClass, newInterface, addClass, addInterface</td>
</tr>
<tr>
<td>file</td>
<td>newClass, newInterface, addClass, addInterface</td>
</tr>
<tr>
<td>type</td>
<td>newMethod, insertMethod, insertCode, addImplements, newField</td>
</tr>
<tr>
<td>class</td>
<td>newConstructor, extractInterface</td>
</tr>
<tr>
<td>method</td>
<td>addParameter, clone, createAdapter, extractFunctionalInterface</td>
</tr>
<tr>
<td>call</td>
<td>clone</td>
</tr>
<tr>
<td>loop</td>
<td>tiling, interchange, unroll</td>
</tr>
<tr>
<td>expression</td>
<td>extractToFile</td>
</tr>
<tr>
<td>field</td>
<td>setInit</td>
</tr>
</tbody>
</table>

These join points, their attributes and actions are the basis for LARA strategies to access the target application source code to restructure the code and add runtime adaptivity. To help the strategies developer the proposed approach includes a library of aspects.

5.2.2 Aspects API

The Kadabra API is a library of LARA aspects and JavaScript classes and functions established as the middleware between LARA runtime adaptivity aspects and the Java API. A runtime adaptivity strategy developer that intends to use the Java API can abstract its use by using the Kadabra API, leveraging the potential between the LARA aspectual functionality and the runtime adaptivity API. Figure 5.1 shows the hierarchy between the different Kadabra API modules.

The Utils, Type and Factory modules provide utility functions that allow one to deal with raw information, such as Java types in plain text, and convert it to useful data, actual Java type in the previous example, or to statically generate code to be injected in the application, avoiding the use of native code. The CpuInfo module is a utility module that allows one to inject code that accesses the Java API to retrieve information from the CPU. The Concurrency module is also a utility module to inject concurrency components in the application. Timer and energy modules are both measurers that can be used to measure execution time and energy consumption at runtime.

The Extractor module is used to expose expressions and more importantly, function calls and automatically build new fields and extract functional interfaces, an important feature when developing strategies for algorithm selection and runtime code generation. The Adapter module allows an easier specification of runtime method adaptation or generation, which is then used by
Programming Runtime Adaptivity in Java with LARA

the Algorithm module that represents the different type of algorithms one may add to the autotuner.

The SimpleExplorer is a module to use when the strategy developer wants more freedom in the definition of the runtime exploration strategy. It does not provide the automatic exploration as an autotuner provides (see, e.g., (Gadioli et al., 2019)) but gives some basic functionality for storing and managing knob configurations. The Measurer, Configs and Algorithm modules are intermediate functionalities the Autotuner module requires to execute. Measurer provides runtime measurers to the autotuner, which can be a Timer or an Energy measurer, while Configs is used to access knobs configurations, their ranges and search schemes. The Autotuner module is used to build autotuners targeting knobs or algorithms. For knobs, the autotuner is actually called as a ControlPoint and for algorithms, it is maintained the Autotuner name.

![Figure 5.1. High-level hierarchical representation of the Kadabra API developed for interaction with the developed Java API.](image)

**5.3 Strategies Specification**

This section describes how one can design a runtime strategy in LARA that uses the Kadabra modules to help in the development of that strategy. With LARA it is easy to select points of interest in the code and apply code transformations to them. The proposed approach takes advantage of this to restructure the source code according to the type of adaptivity one intends to add to the target application. In this stage, the join points actions and attributes and the utility modules available in the Kadabra API are used. These sections show how to specify adaptivity strategies following the features available in the approach to restructure the code based on the type of target adaptivity and the use of the Kadabra API to deliver adaptivity.
5.3.1 Runtime Adaptivity with Knobs

When the target of adaptivity is a set of software and/or system parameters, one can define strategies in LARA to expose those target parameters and insert the code to manage the tuning criteria. Figure 5.2 shows a LARA strategy for tuning k based on the same strategy defined in Figure 4.1. The implementation is very close to the expected strategy, where we select the target knob (line 2), the update point (line 3) and relevant runtime information (line 4). It is possible to see that pure Java code can be directly injected in the application (lines 6 to 13). The native code can be parameterized with compile-time information using the double square brackets (“[[...]]”) (lines 8 to 10). The resulting weaved code considers the actual value of the information accessed inside the square brackets.

```java
aspectdef TuneKNN
  kNN: select class("kNN") end //the adaptation point
classifyCalls: select method.call("classify") end //the update point
numClassesParam: select method.param("numClasses") end //the scenario data
apply to classifyCalls: kNN:numClassesParam
apply before % (insert Java code that executes the runtime tuning
switch([[$numClasses]]){
  case crit1: [[$kNN]].setK(3); break; //the current number of classes
  case crit2: [[$kNN]].setK(5); break;
  case crit3: [[$kNN]].setK(7); break;
  ...
}%;
end execute
end foreach
```

Figure 5.2. LARA aspect implementing the strategy depicted in Figure 4.1 that tunes the number of neighbors (k) to consider on kNN, based on runtime criteria.

The use of SimpleExploration is ideal for explorations in which one intends more freedom to specify the autotuning code that should execute, following an event-based approach. This means that it is an approach that requires more native code written by the strategy developer and they must write the assignments to the target knob and write specifically when the adaptation should stop. Furthermore, unless the developer writes more code and adds more variables, it does not have a way of maintaining explorations for each different matrices sizes that may occur during execution.

Figure 5.3 shows an aspect using the SimpleExploration approach to control the block size of a tiled loop and find the best performing block size. First, the strategy calls (in line 5) the LoopTile aspect (defined in lines 21 to 29) that applies loop tile transformation over the outermost loop of a target method (line 27), which immediately exposes the block size as a field. Then, all the calls to the method in which loop tile transformation was performed are selected (line 7) and for each, a SimpleExploration is built based on the target field reference and the number of times a current configuration is sampled (line 9). Lines 10 to 16 defines the search scheme (line11) and when the exploration should stop (lines 12 to 15). The search scheme defined in this exploration is a simple approach in which it starts by the default block size and for each new configuration, it increments a step value on the previous configuration until a given limit is achieved. The exploration, i.e. the update and the measurements, is performed around the target calls (line 17).
Programming Runtime Adaptivity in Java with LARA

```java
import kadabra.SimpleExploration;

// Apply loop tiling to a target method and build an exploration for its calls
aspectdef AdaptWithTile
input targetMethod, defaultBlockSize = 32, step, numRuns, limit end
tiling = call TileLoop(targetMethod, defaultBlockSize); // apply loop tiling
var blockRef = tiling.block; // get a reference to the field of the tiling block
select method.call{targetMethod} end
apply // build a new exploration for that block
   exp = call SimpleExploration(blockRef, numRuns);
   exp.onNewVersion(%{
      // every "numRuns" calls a new version is requested
      [[blockRef]] = [[blockRef]] + [[step]]; // increase block size with step
      if([[blockRef]] > [[limit]]){
         // check if limit was reached
         [[blockRef]] = [[exp.$bestVersion]]; // update block ref to use best
         [[exp.stop]] // and stop the exploration
      }
   }%);
   exp.aroundStmt($call); // measure time around method call end
end

// aspect to do loop tiling to the outermost loop of a target method
aspectdef TileLoop
input targetMethod, defaultBlockSize end
output $block end
// select the outermost loop of the target method
select method{targetMethod}.loop{isoutermost==true, type="for"} end
apply // apply loop tiling and get a reference to the exposed block
   $block = $loop.exec tile(defaultBlockSize);
end
```

Figure 5.3. Loop tiling tuning strategy defined in LARA that samples block sizes from 32 to a limit given as input matrix height, in powers of 2.

### 5.3.2 Runtime Algorithm Selection

Algorithm selection is based on extracting method invocations to fields. The proposed approach for runtime generation takes advantage of Java 8 functional programming (Warburton, 2014) using the ExtractFunctionalInterface and ExtractToField actions. With these two actions, one can control method invocations as if they were simply parameters. Based on a target method ExtractFunctionalInterface generates a functional interface, which contains a method with the same signature as the invoked method. Then, using this new interface, ExtractToField generates a new field inside a class and initializes it as default with the target method of the invocation. The direct invocation of the method is then replaced with an invocation over the newly added field.

Figure 5.4 shows the use of this functionality and the injection of the adaptation code that based on the first input argument of the target function call (i.e., the scenario) it selects the corresponding algorithm. It starts by selecting calls to a target method and the first argument of the call (line 4). Then, a functional interface is extracted from the calls, based on the target method signature, a new field with that interface type is introduced and the call is replaced with a new call to the new field (line 6). This allows the call to be easily tuned at runtime. Lines 7 and 8 gather the algorithms to be used and map the possible known scenarios to the corresponding algorithm. Lines 9 to 16 insert code to be executed before the function call. This code manages the algorithms to use, where the selection is done by the first input argument of the call (line 10).
5.3.3 Runtime Generation

The proposed approach provides aspects that facilitate the inclusion of method generators within an application with minimal effort, whether for method redefinition or to dynamically define new functional methods. If the intention is to replace an existing method, then the ClassAdapter aspect can be used in LARA code, providing the name of the template and a reference to the target method. If instead one intends to add new methods based on a functional interface, i.e., functional methods, the ClassGenerator aspect can be used, providing the template and the functional interface.

Figure 5.5 shows an example of an aspect using the ClassAdapter aspect. Based on the given template and target method, this aspect generates and inserts in the application the method required to adapt the target method at runtime (line 2). The invocation of that method is injected before a target statement (lines 3 to 5). The adapt method of the ClassAdapter (line 4) is used to generate the new code at runtime and replace the existing bytecodes of the target method. This method accepts as arguments the same inputs as the ones required by the template.

```
1 // Define a functional method generator
2 adapter = new ClassAdapter('template',$method);
3 $statement.insert before %{ [[adapter.adapt(param1,param2)]] }%
```

Figure 5.5. Example of a dynamic method adaptation with a template that requires two parameters.

The ClassGenerator aspect can be used in a similar way with the difference that now the invocation of the generator, instead of replacing the target method, it will return a reference to a new class that contains the generated method. This approach is useful to maintain multiple versions of an algorithm and to avoid the regeneration of algorithms. Figure 5.6 shows an aspect that dynamically generates specialized versions for the target function call, keeping a map with
the generated versions. The strategy starts by extracting the function call to a field (line 5) and adding a new map that maps an input value to a specific generated version (line 7). In line 9 a `ClassGenerator` is inserted in the application based on a template and the type of the new field. Lines 10 to 17 inserts code before the target function calls to dynamically generate new versions. If the map of algorithms already contains a version for the given scenario (line 11), then the field is assigned with the mapped version (line 15). Otherwise, a new version has to be generated. The generate method of `ClassGenerator` accepts as input the same argument as the template expects and it outputs the generated class. This class implements the same interface as the new field and so it can be assigned to it (line 12). This new version is stored in the map with the current scenario as the key (line 13).

```
aspectdef MethodGenerator
input fCallName end
select class.call{fCallName}.arg{0} end
apply
$funcField = $call.exec extractToField(); //extract call to a field
//introduce a map mapping scenario→generated algorithm
$map = $class.exec newField (’algorithms’, Map.of(Types.int,$funcField.type))
// Define a functional method generator
generator = new ClassGenerator(’template’,$funcField.type);
$call.insert before %
  if(!algorithms.contains([[$arg]])){ //if new scenario
    [[$funcField]] = [generator.generate($arg)]; //generate new version
    algorithms.put([[$arg]], [[$funcField]]); //and store in map
  }
else{
  [[$funcField]] = algorithms.get([[$arg]]); //use existing version
}
end
end
```

Figure 5.6. Example of a dynamic selection of the algorithm to use. When an algorithm does not exist for a given scenario, a new class is dynamically generated with a template accepting one argument.

Both `ClassAdapter` and `ClassGenerator` are used in the runtime algorithm selection and for the autotuning features, providing these features with more adaptivity potential. It provides new versions for the algorithm selection to be sampled and avoids having all possible versions precompiled. In explorations using extensive search schemes, new explorations take advantage of `ClassGenerator` for having previously generated versions from other explorations.

### 5.3.4 Autotuning

In order to achieve a more complete and more automated exploration approach, and as an alternative to the SimpleExploration, one may use ControlPoint or Autotuning features. ControlPoint is an API that requires less native code (almost none) written by the developer and allows a different exploration for given data input. The use of this API is advantageous when the developer intends to use an exploration approach that manages the explored version and automatically store the best configuration.

Figure 5.7 shows a strategy for adapting a parameter by means of an exposed block of a tiled loop. The example starts by preparing the target loop with the tile action, using the provided block field reference (line 2). Then, it defines the configuration for the target knob, specifying that this knob can have values between 32 and 2048 (line 4) and the search configuration is the default (line 5), i.e., it explores all possible values within the knob range. A control point for
the block is defined in line 7, which is responsible to explore the values the knob can take, measure the execution time and use the best (found) version when the exploration is done. These steps so far define the adaptation point. Line 9 determines how the block is updated and when measurements are in order, using the updateAndMeasure method that injects the update and measuring code around a provided join point (a loop in the example). These two steps can be used separately using the methods update and measure for different target join points.

```java
//apply loop tiling
$blockRef = $loop.exec tile(defaultValue);
//define a knob: specify range and search config.
knobRange = new Range(32, 2048);
knobConfig = Configs.defaultScheme($blockRef, knobRange);
//associate a control point to the block
cp = new ControlPoint(knobConfig);
//insert code around the loop
cp.updateAndMeasure($loop);
```

Figure 5.7. Example of an aspect that tiles a loop and dynamically searches for the best tile size by means of a ControlPoint with an exhaustive (default) exploration.

The Kadabra API also provides an exploration of different type of algorithms by means of a runtime adaptation manager called Autotuner, which allows the management of a list of candidate algorithms/versions. The Autotuner provides new versions to the application to execute while the application feedbacks measurements. It is configured with the list of algorithm candidates, the type of monitor to use and the locations where adaptation, measurement and update should occur. This API can be used in two modes: exploration and best mode. In exploration mode, the Autotuner provides new versions to explore while waiting for new measurements for those versions. In best mode, the Autotuner selects the best performing version based on the measurements.

The current version of the Autotuner accepts three types of algorithms: simple, with knobs and generative. Simple algorithms do not require any preparation to execute. Algorithms containing knobs are similar to knobs specified as control points, with the difference that the knobs related to that algorithm only have influence in the application if the specified algorithm is executed. Similar to a ControlPoint, these algorithms require a knob configuration. Generative algorithms use the same functionality as the program specialization, where a template is given for the code generation and the template requires input arguments for the generation. The input arguments can be specified with constant values, a runtime parameter or a configuration can be provided to explore different ad-hoc versions.

Figure 5.8 shows a possible LARA aspect implementation of the recipe presented in Section 3.1, using Kadabra API features. The strategy considers a set of pre-compiled algorithms (lines 2 to 4) and an algorithm with an associated knob (lines 5 and 6). Calls to a target method and an input argument are selected in line 8 to mark the location where code restructuring shall occur. Line 11 uses a Kadabra API aspect that generates a functional interface based on the method signature and replaces the call with an invocation to a generated field to which the current target method is assigned as the default value. This aspect does not change program behavior, but exposes the method call so it can be dynamically resolved. Lines 13 to 25 build the Autotuner by adding a few components (a set of simple algorithms, one algorithm with a knob, two generative algorithms, and a time monitor). The algorithm with a knob uses a knob that will be explored with a full exploration configuration (lines 14 and 15). The generative algorithms use two templates, each generating different specialized versions based on the input argument. The last statements
Programming Runtime Adaptivity in Java with LARA

(lines 27 to 29) inject code to control the runtime adaptivity. The adapt method sets the adaptation point, while the updateBefore method sets the update point and the update point is implicitly considered as the trigger point. Finally, the monitor method is used to specify the measurement point, the code statement/region that needs to be monitored.

```java
1// (1) Expose relevant information
2simpleAlgs = []; // gather all methods
3simpleAlgs.push(GetMethod('alg1'));
4...
5algWithKnob = GetMethod('algWithKnob');
6knob = GetField('ParCountingSort', 'nThreads');

7// (1) select calls to sort
8select call(targetFCall).arg end
9apply
10// (2) Build functional interface and replace call
11extracted = ExtractFunctionalInterface($call);
12var interfaceType = extracted.$interface.type;

13// (5) Autotuner setup
14knobRange  = new Range(2, 32, 2);
15knobConfig = Config.defaultScheme(knob, knobRange);
16autotuner  = new Autotuner(interfaceType);
17// (3) list algorithms
18for(var alg of simpleAlgs){
19  autotuner.addAlgorithm(alg);
20}
21autotuner.addAlgorithmWithKnob(algWithKnob, knobConfig);
22  .addGenerativeAlg('Template1.tpl', $arg.length)
23  .addGenerativeAlg('Template2.tpl', $arg.length)
24  .addMonitor(Monitor.timer); // (4) add monitor
25  .build();

26// (6) inject autotuning features
27autotuner.adapt(extracted.$field);
28autotuner.updateBefore($call);
29autotuner.monitor($call);
30end
```

Figure 5.8. LARA strategy that performs algorithm exploration at runtime using multiple algorithms.

5.4 Summary

This chapter presented the proposed approach on leveraging runtime adaptivity in an application by designing strategies with the LARA language coupled with a Java-to-Java weaver and a Java API with runtime adaptivity features.

It is discussed the developed Java API, the implementation of the Java-to-Java weaver, named Kadabra, and the set of LARA modules that were developed to interface LARA with the Java API. Then, the definition of runtime adaptivity strategies by means of the LARA language and those modules were discussed and exemplified.
6 Case studies

Contents

6.1 Matrix Multiplication .................................................................................................................. 118
  6.1.1 Prepare for Tiling Exploration ............................................................................................... 119
  6.1.2 Loop Tiling with fixed size .................................................................................................... 120
  6.1.3 Explore Different Blocks Incrementally ................................................................................. 121
  6.1.4 Explore Neighbors Around a Block ....................................................................................... 123
  6.1.5 Concurrent Exploration ......................................................................................................... 125
  6.1.6 Random Block Sizes Exploration ......................................................................................... 128

6.2 FIR ........................................................................................................................................... 128
  6.2.1 Runtime specialization ............................................................................................................ 129
  6.2.2 Runtime Exploration of Unroll Factors .................................................................................. 130
  6.2.3 Autotuner With Specialization ............................................................................................... 130
  6.2.4 Autotuner to Explore Unroll Factors ..................................................................................... 131

6.3 Median Smooth ......................................................................................................................... 131
  6.3.1 Extract Function Call .............................................................................................................. 132
  6.3.2 Autotuner with Predefined Algorithms ............................................................................... 133
  6.3.3 Autotuner with Specialized Sorting Network and Counting Sort ....................................... 134
  6.3.4 Autotuner with Generative Algorithms ............................................................................... 135
  6.3.5 Switch to Pre-Selected Algorithms ....................................................................................... 135

6.4 Summary .................................................................................................................................. 136
Case studies

This chapter presents three case studies in which adaptation opportunities where found. It is presented the developed runtime adaptivity strategies over the application kernels and shown how the development of these strategies are incremental and can be reused for other strategies.

6.1 Matrix Multiplication

The first case study is the matrix multiplication task presented in Figure 6.1, a common implementation of the multiplication algorithm that naively iterates over two matrices without considering the best iteration order.

```java
public static int[][] mult(int[][] a, int[][] b) {
    int rowsInA = a.length;
    int columnsInA = a[0].length; // == rows in B
    int columnsInB = b[0].length;
    int[][] c = new int[rowsInA][columnsInB];
    for (int i = 0; i < rowsInA; i++) {
        for (int j = 0; j < columnsInB; j++) {
            for (int k = 0; k < columnsInA; k++) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    return c;
}
```

Figure 6.1. A Java implementation of a matrix multiplication operation.

The adaptation proposed for this algorithm is to, first, study a more adequate loop order that results in the best cache use and to specify strategies to dynamically search for the best tile block size for the loops of matrix multiplications to which loop tiling was applied prior to the execution. It is also intended to observe the adaptation overhead between a sequential and a concurrent adaptation.

All the strategies have an initial static adaptation, in which, and based on profiling over the loops, it is applied loop interchange followed by loop tiling to the two outermost loops, using the same tile block variable to control the block size.

The search of the best tile is based on simple techniques for autotuning:

- full exploration from minimal tile to a specific maximum (e.g., one of the matrices dimensions);
- exploration with random values;
- starting with L2 cache size, search below and above this value.

As the adaptation strategies used require simple computations, the overhead is expected to be negligible. The objective here is to observe, between the different strategies, which one achieves faster the (approximated) best block size for the current execution providing the best cache performance (Lam et al., 1991). Furthermore, the evaluation includes some of the autotuning strategies written in different ways to demonstrate that the proposed approach allows the strategy specification in different fashions and with different APIs and the pros and cons of using some of the developed API features.
A previous profiling analysis identified the best loop interchange between the three loops iterating the two matrices prior to the strategies definition. The experiments include the possibility to use a template generator, or to have multiple versions generated at compile-time, and examine all loop interchange possibilities at runtime. However, since the intention here is to find the best tile size, it only considers the interchange that provides the best profiling results at compile-time.

Figure 6.2 shows the average time of executing the matrix multiplication with squared matrices of size 1024 by 1024. The profile showed that interchanging loop $j$ and $k$ ($ikj$ in the chart) provides 8.5× speedup against the original version (see Figure 6.1). Another possibility is interchanging loops $i$ and $j$ with $k$ ($kij$ in the chart). These experiments, however, show that the use of loop interchange between $j$ and $k$ is the one providing better performance.

<table>
<thead>
<tr>
<th>Interchange</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ijk$</td>
<td>9.19</td>
</tr>
<tr>
<td>$ikj$</td>
<td>1.08</td>
</tr>
<tr>
<td>$jik$</td>
<td>8.37</td>
</tr>
<tr>
<td>$jki$</td>
<td>9.91</td>
</tr>
<tr>
<td>$kij$</td>
<td>1.18</td>
</tr>
<tr>
<td>$kji$</td>
<td>9.57</td>
</tr>
</tbody>
</table>

Figure 6.2. Results of all combinations of loop interchange for a matrix $1024 \times 1024$ when running in a PC with an Intel® Quad Core™ i5 650 @ 3.20GHz CPU and 8 GB DDRAM.

### 6.1.1 Prepare for Tiling Exploration

Knowing the type of loop interchange the strategy must do, the code is now prepared for the runtime adaptation. As a first approach, it is defined a strategy that applies two compile-time optimizations that improve the matrix multiplication in a static manner, i.e., no runtime adaptation is defined.

The code transformations applied in this phase are loop interchange and scalar replacement. Figure 6.3 depicts this first strategy. Loop interchange can be accomplished with the *Kadabra* action *interchange* of the join point *Loop*. This action accepts as input the loop with which the current loop has to switch. The aspect defined in Figure 6.3 uses loop interchange in loops $j$ and $k$ (lines 2-5) by selecting the two nested loops inside method *mult* (line 2) and then apply the interchange action (line 4).

Having the $j$ and $k$ loops of Figure 6.1 switched, the expression $a[1][k]$ in the assignment in line 9 does not change in the innermost loop and can be extracted into a temporary scalar variable before that innermost loop. This avoids excessive array accesses done (all iterations of loop $k$) by accessing the array only once per $j$ iteration. Lines 7-12 in Figure 6.3 depict a strategy to extract the target expression. The select in line 7 is a fine-grained selection, that selects the left-hand side of the multiplication (the target expression) inside the right-hand side of the target assignment. The *extract* action is used to transfer the expression to a scalar variable, providing a name for the variable and the new location (line 11).
Case studies

### 6.1.2 Loop Tiling With Fixed Size

This second strategy is the following step to include runtime adaptivity in the execution regarding the use of loop tiling. This action is applied over the two inner loops \((k \text{ and } j)\) and wraps the outermost loop. Figure 6.4 depicts the loop tiling extension to the strategy defined in Figure 6.3, where it starts by selecting the loops that are not outermost (line 1) and separately select the outermost loop (line 2). Then it applies to the joined selection the loop tiling over the inner loops (line 3), taking as reference the name of the block variable and the statement that the tiling loop must wrap, which is \(\$outer\) loop.

```
$inner$inner=loop(isOutermost == false) end //OR
controlVar != 1
$outLoop$outer=loop(isOutermost == true ) end //OR
controlVar == 1
apply to innerloops::outermost
$inner$inner=loop('block', defaultBlockSize, $outer); //tiling the inner loops
```

- Figure 6.4. A weaving-time strategy that tiles the two innermost loops, defining the tiling loops around the outermost loop.

The final version of \(\text{mult}\) is depicted in Figure 6.5, considering a default tiling block of size 256. Although the current strategy only specifies a default value for the tile size (i.e., no runtime

```
public static int[][] mult(int[][] a, int[][] b) {
    int rowsInA = a.length;
    int columnsInA = a[0].length; // == rows in B
    int columnsInB = b[0].length;
    int[][] c = new int[rowsInA][columnsInB];
    for (int i = 0; i < rowsInA; i++) {
        for (int k = 0; k < columnsInA; k++) {
            int temp = a[i][k];
            for (int j = 0; j < columnsInB; j++) {
                c[i][j] += temp * b[k][j];
            }
        }
    }
    return c;
}
```

- (b) Matrix multiplication after applying loop interchange \((j \leftrightarrow k)\) and scalar replacement figures 6.3. Weaving-time strategy (a) that interchanges loops \(j\) and \(k\) and extracts \(a[i][k]\) expression (see Figure 6.1, line 9) to a scalar variable and the resulting weaved code (b).
adaptness will be performed), the exposed block field allows the strategy to dynamically change the size of the block, as it is illustrated in the following strategies.

```java
public static int block = 256;

public static int[][] mult(int[][] a, int[][] b) {
    int rowsInA = a.length;
    int columnsInA = a[0].length;
    int columnsInB = b[0].length;
    int[][] c = new int[rowsInA][columnsInB];
    for (int kBlock = 0; kBlock < columnsInA; kBlock += block) {
        for (int jBlock = 0; jBlock < columnsInB; jBlock += block) {
            for (int i = 0; i < rowsInA; i++) {
                for (int k = kBlock; k < Math.min(columnsInA, kBlock + block); k++) {
                    int temp = a[i][k];
                    for (int j = jBlock; j < Math.min(columnsInB, jBlock + block); j++) {
                        c[i][j] += temp * (b[k][j]);
                    }
                }
            }
        }
    }
    return c;
}
```

Figure 6.5. Matrix multiplication method of Figure 6.1 after applying loop interchange (j↔k), scalar replacement and loop tiling over the two inner loops. The changes made over the code are highlighted.

### 6.1.3 Explore Different Blocks Incrementally

This strategy uses the exposed tiling block to explore and find the best block size for a given input matrix size. The search is accomplished by exploring different sizes of blocks, starting from a block of size 32 and incrementing it in powers of 2 until the height of the left-hand side matrix is reached. The search and adaptation are done sequentially to the application.

Figure 6.6 depicts this strategy, starting by invoking the previous strategy (line 3) and add the adaptation phase in each call to method `mult` (lines 5 to 16). The SimpleExploration aspect API is used in Figure 6.6 to deal with the exploration phase, which in this case only requires the code (lines 8 to 13) that offers new values to explore (line 9) and decides when to stop (lines 10 to 13). The next version to sample is essentially a left shift that gives the next power of 2. The stopping criterion are defined by the height of the left-hand side matrix. Finally, to measure the versions the strategy inserts code around the `mult` call by invoking the `aroundStmt` method of SimpleExploration, which inserts the necessary code to adapt the block, to time `mult` execution and to update the measurements.
Case studies

```java
import kadabra.SimpleExploration;
aspectdef AdaptWithTile
input defaultBlockSize = 32, numRuns end
call tiling: Strategy2(defaultBlockSize); //mmult preparation with loop tiling
var blockRef = tiling.block;
select class.method.call('mult').arg{index==0} end
apply
call exp: SimpleExploration('int', blockRef, numRuns);
exp.onNewVersion(%
[[blockRef]] = [[blockRef]] << 1; //increase block size with a left shift
if([[blockRef]] > [[arg.length]]){
    [[blockRef]] = [[exp.$bestVersion]]; //update block ref to use best
    [[exp.stop]] = true; //and stop the exploration
}
exp.aroundStmt($call); //measure time around mmult call
end end
```

Figure 6.6. Tiling strategy that explores block sizes from 32 to matrix height, in powers of 2.

Figure 6.7 depicts a snippet of the resulting code of injecting the adaptation around a call to method `mult` (line 17), simplified for space and readability. The injected code is bigger and more complex compared to the actual native code the user wrote (lines 8 to 12).

```java
If(adapt) { // adapt and start timer
    ... // reset variables if first run
    if(numRuns == 0){
        if(currTime < bestTime){ //if current version is better than the best version
            bestVersion = block; // 'block' is given by 'blockRef' in the aspect
            bestTime = currTime;
        }
        block = block << 1;
        if(block > matrixA.length){
            block = bestVersion;
            adapt = false;
        }
        ...//reset counters and timers for the next configuration;
    }
    timer.start();
}
int[] result = mult(matrixA, matrixB);
if (adapt){ // stop timer and update total sampling time
    timer.stop();
    if (!warmup) {
        currTime += timer.getTime();
        numRuns--;
    }
    ...
}
```

Figure 6.7. A snippet of the code that is injected around `mult` call for the exploration.

The use of `SimpleExploration` is ideal for explorations in which one intends more freedom to specify the autotuning code to execute. Meaning that the strategy developer needs to write more native code, to write the assignments to the target knob, to write specifically when the adaptation should stop, etc. Furthermore, unless the developer writes more code and adds more variables, he/she does not have a way of maintaining explorations for each different matrices sizes that may occur during execution, only maintaining the version providing the best execution time.
As an alternative, one may use the knob autotuning API named ControlPoint, which requires less native code written by the developer and can maintain different explorations, such as one per matrix size. Figure 6.8 shows the use of this API. The strategy starts by specifying the configuration of the target knob, for which it defines the exploration range and the type of increment that is intended to use (lines 9 and 10). Then, it uses the ControlPoint builder (lines 12 to 14) to create a ControlPoint for this knob configuration, stipulating that it is intended an exploration for different matrix sizes ($\text{arg}.\text{length}$), each version monitored numRuns times and the type of monitor to use is the average execution time.

```plaintext
aspectdef AdaptWithTile
  input defaultBlockSize = 32, numRuns end
  call tiling: Strategy2(defaultBlockSize); //mmult preparation with loop tiling
  var blockRef = tiling.block;
  select class.method.call('mult').arg(index==0) end
  apply
    //specify block range and search config
    knobRange = new IntegerRange(32, $arg.length, 'v-> v << 1'); //left shifts
    knobConfig = Config.default(blockRef, knobRange);
    //associate a control point to the block
    var cp = new ControlPoint(knobConfig, $arg.length, numRuns)
    .setMeasurer(Measurer.avgTime()) //measure execution time
    .build;
    //insert code around the loop
    cp.updateAndMeasure($call)
end
```

Figure 6.8. Same tiling strategy as Figure 6.6 using ControlPoint instead.

### 6.1.4 Explore Neighbors Around a Block

The “Explore Neighbors around a block” strategy tries to improve Strategy 3 by reducing the number of configuration to explore. The idea is to find an approximate best block size by starting from a specific size and looking at the left and right, power 2, values until the performance starts to degrade.

Figure 6.9 depicts a strategy written with the SimpleExploration API, using half of the L2 cache size as the starting block and searching its surrounding, power of 2, neighbors. The CpuInfo aspect (line 4) provides functionalities that retrieve runtime information about the processor, which in this case is used to get the L2 cache size at runtime (lines 5, 17 and 29). The main difference to Strategy 3 is the code to provide a new configuration, as well as the stopping criteria. Each block size is sampled numRuns times, without warmups, and if the current configuration improves the execution reduces the block size to half its value. If it does not improve, then it starts looking to values higher than the L2 cache size. If a configuration with an augmented block does not improve, the adaptation stops and uses the best block so far. The strategy also adds an additional boolean field, shiftBack, to control the direction of the exploration.
Figure 6.9. Tiling strategy that explores block sizes starting from half of the L2 cache size. The strategy starts by sampling lower values for the block and searches upwards when performance degrades.

Figure 6.10 shows a code excerpt of the injected code where it is shown the use of `CpuInfo` library. In this strategy, more native code had to be written to control when to increase/decrease the block value, new fields add to be injected simply to control the direction of the exploration.
public static int block = weaver.kadabra.system.CpuInfo.getL2CacheSize() << 1;
...
if (improved) { // if best version was updated
    if (shiftBack) { // if sampling lower values
        block = block >> 1; // e.g. 256 -> 128
    }
    if (block < 2) {
        shiftBack = false; // shiftUp
        block = kadabra.system.CpuInfo.getL2CacheSize(); // from L2 cache
    }
} else {
    block = block << 1; // e.g. 128 -> 256
    if (block > matrixA.length) { // exploration achieved limit
        block = bestVersion; // use best version
        adapt = false; // and stop the exploration
    }
}

Figure 6.10. A snippet of the code that is injected before mutl call. The CpuInfo.l2 used in the aspect was translated to the CpuInfo.getL2CacheSize() method invocation.

Again, for a simpler approach can use ControlPoint with the same exploration scheme of Figure 6.9 as depicted in Figure 6.11. Here, it is used the same strategy defined in Figure 6.8 with the main difference being the definition of the knob range and the configuration. The knob is now defined as starting with the L2 cache size (line 12) and both ascending and descending functions are now required (line 13), and the exploration is now defined as “around” (line 14).

import kadabra.system.CpuInfo;
aspectdef AdaptWithTile
input numRuns end
call cpu: CpuInfo(); // provides functions to retrieve cpu info
var l2Size = cpu.l2.orElse(256); // L2 cache size, if not accessible use 256
call tiling: Strategy2(l2Size << 1); // loop tiling with “half L2 cache size”
var blockRef = tiling.block;
select class.method.call('mult').arg{index==0}
apply
    // specify block range and search config
    blockRange = new IntegerRange(2, $arg.length, l2Size,
        'v-> v >> 1', 'v-> v << 1');
    knobConfig = Config.around(blockRef, knobRange);
    //associate a control point to the block
    var cp = new ControlPoint(knobConfig, numRuns)
        .setMeasurer(Measurer.avgTime()) //measure execution time
        .build;
    //insert code around the loop
    cp.updateAndMeasure($call);
end
end

Figure 6.11. Strategy using the ControlPoint API with the same criteria as in Figure 6.9.

6.1.5 Concurrent Exploration

This strategy executes the time measurements analysis and decisions of new version concurrently to the program. This strategy requires a different approach of “Explore Neighbors around a Block” strategy (see section 6.1.4) since it needs program/strategy synchronization. Hence, the new fields used for sampling use a more secure type for concurrency, i.e., atomic types11. Furthermore, it

11 See https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html
uses a communication channel to transfer multiple timing measurements to the concurrent
strategy and a simple Atomic type for the sampling version.

Figure 6.12 partially depicts the new code that is necessary for the concurrent strategy. In
line 3 one can see the addition of a concurrent channel. A channel requires a tuple (id, value).
This way, when timestamps are reported, the strategy knows which version was executed. Line 4
shows the creation of a new thread that receives not only the class owning this thread but also
where the thread should start and stop, which in this case is inside the main method. Additionally,
the strategy adds a utility method, named measure, which is the one expecting timestamps from
the channel. The arguments of this method are: the number of times to sample before reporting
the average and the id the current measurement is expecting. Timestamps with a different id are
discarded from the measurement. Using the id avoids the measurement of versions that are not
the expected, such as the execution of previous configurations. This method halts its execution
while it is expecting a new timestamp (line 15).

```java
import kadabra.Concurrent;
...
call channel: NewChannel($class, 'int', 'long', $channelSize); // for measurements

call thread: NewThread($class, $mainMethod); // (owner, starter)
call config: NewAtomic($class, 'Integer', 'block'); // optional sampling
call best: NewAtomic($class, 'int', 'bestBlock', $defaultBlock); // best version
call adapt: NewAtomic($class, 'bool', 'adapt', false); // adaptation on/off
call timer: NanoTimer($class); // for timing the execution

$class.$exec insertMethod(%{ // inject utility method to measure a version
    private static long measure(int numSamples, int id){
        int count = 0;
        long accTime = 0;
        while(count < numSamples & & [thread.running]){ // halts until available
            if($channel.ValueType].get($channel.take());
                continue; // Ignore measurements with different id
            }
            count++;
            accTime += $channel.getValue();
        }
        return accTime/count;
    }
}
...}
```

Figure 6.12. Definition of the elements necessary for the concurrent execution. This is part of the code
assembling phase. The thread is started, and terminated, in the main method, given by the join point
$mainMethod.

Now it is needed the code that executes in the program side, depicted in Figure 6.13 and the
one executing in the thread side, defined in Figure 6.14. The code executing in the program
verifies if the execution is in sampling mode (Figure 6.13, line 4) and, if so, gets the sampling
configuration if available, and starts the timer before calling mult method and stops the timer
after the execution to report this timestamp to the channel. The code executing the thread is
responsible to define the new configurations and to measure the current configuration by means
of the injected method measure (Figure 6.14, lines 3 and 20). The thread stops when the
exploration is finished, i.e., when the approximate best version is found. The assignment of atomic
variables is not done using the assign statement but with the invocation of the set method;
similarly, the get method is used to read an atomic variable.
select class.method.call('mult') end

$call.insert before{% if [[adapt]]{ //if exploring versions
    [blockRef] = [[config.getOrElse(best.get)]]; //use config. or else best
    [timer.start]] //time mult call
}}%

$call.insert after{% if [[adapt]]{ //stop timing mult call
    [[channel.offer(blockRef,timer.getTime)]]; //offer the measured time
}}%

Figure 6.13. Inserting code to be executed by the program, i.e., acquire configurations and provide time measurements to the channel.

thread.setCode(%{
    // defining the adaptation code
    int targetVersion = [[cpu.l2]]; // start with original version
    long bestTime = measure([[numRuns]], [[targetVersion]]); // measure original
    boolean shiftBack = true;
    [[adapt.set(true)]]; // activate adaptation
    while([[thread.running]]){
        if(shiftBack){
            targetVersion = targetVersion >> 1;
            if(targetVersion < 2){
                shiftBack = false;
                targetVersion = [[cpu.l2]];
            }
        }else{
            targetVersion = targetVersion << 1;
            if(targetVersion > [[limit]]){
                break; // exploration achieved limit
            }
        }
    }
    [[config.set(targetVersion)]]; // set sampling configuration
    long sampleTime = measure([[numRuns]], [[targetVersion]]); // measure config.
    if(sampleTime < bestTime){ // if better update best
        bestTime = sampleTime;
        [[best.set(targetVersion)]];
    }else if(shiftBack){ // shift forward
        targetVersion = [[cpu.l2]];
        shiftBack = false;
    }else{
        break; // stop adaptation
    }
}]

[[blockRef]] = [[best.get]]; // at the end use best version
[[adapt.set(false)]]; // and deactivate adaptation

Figure 6.14. Adaptation code to be executed in a concurrent thread.

The use of ControlPoint makes even easier to use concurrency, as the only change required is to add a simple setter in the ControlPointBuilder that sets the concurrency to true, just as follow:

var cp = new ControlPointBuilder(knobConfig, numRuns)
    .setMeasurer(Measurer.avgTime())
    .setConcurrent(true) // measure/generate new versions concurrently
    .build();
This concurrent adaptivity allows the program to continue its execution without waiting for the measurements to be processed and for the generation of new configurations. While the adaptivity strategy executes the measurements processing and the selection of new configuration to sample, the application will continue to use the current version.

6.1.6 Random Block Sizes Exploration

The “Random Block Sizes Exploration” strategy is similar to the “Concurrent Exploration” strategy. The difference resides in the adaptation code, where a set of random values are sampled instead of using a neighbor search approach. Moreover, the previous strategy continues executing the current configuration while analyzing measurements and providing a new version concurrently. Consider the moment in which the measuring method stops measuring and returns the average time for a given version. While the strategy is comparing and updating the new version, select/calculate a new configuration and assigning it, the program is still executing the “old” configuration. Instead of executing the sampling configuration, the program could use the best version while it waits for a new configuration, even if it is still in sampling mode. One can improve the strategy by nullifying the sampling configuration as soon as the measurements stop, forcing the adaptation to execute the current best version. Figure 6.15 depicts the new code for the thread. It defines an array of random values (line 3) and samples each value as a block (lines 4 to 13), removing the configuration as soon as the measurement stops (line 8).

```
thread.setCode(%{
int[] configs = {256, 1024, 32, 512, 128}; // random configurations
for(int i = 0; i < configs.length && [thread.running]; i++){
    targetVersion = configs[i];
    [[config.set(targetVersion)]];
    long currTime = measure([[numRuns]], [[targetVersion]]); // measure config.
    [[config.set(null)]]; // remove configuration
    if(currTime < bestTime){
        bestTime = currTime;
        [[best.set(targetVersion)]]; // update best
    }
}[[blockRef]] = [[best.get]]; // at the end use best version
...%);
```

Figure 6.15. Strategy snippet to explore random block sizes. This code replaces the adaptation body of Figure 6.14.

As for the ControlPoint approach, specifying the configuration as random is enough to have a randomized exploration.

6.2 FIR

The FIR (Finite Impulse Response) (Rabiner and Gold, 1975), with a code example depicted in Figure 6.16, is a filter used in digital signal processing, using a coefficients window (henceforth referred to as window) of size N to process a given sample, having as reference the previous N-1 samples. As input, the method receives an array of samples for processing and an array, of size N, with the coefficients given for each position of the window. In lines 7 and 8, the window is
used to process the current signal with the previous signals. The window size and values do not change inside the given method.

```java
public static int[] fir(int[] x, int[] c) {
    int[] y = new int[x.length];
    int M = x.length;
    int N = c.length;
    for (int j = N - 1; j < M; j++) {
        int output = 0;
        for (int i = 0; i < N; i++) {
            output += c[i] * x[j - i];
        }
        y[j] = output;
    }
    return y;
}
```

Figure 6.16. The FIR method in Java code.

For this method, it is considered herein two types of adaptation that may improve the FIR execution time. The first type is based on runtime specialization, as the inner-loop can be specialized for the given coefficients window (argument ‘c’ for the fir method), applying optimizations with factors depending on the window size and the corresponding coefficients. As this type of specialization may not suffice in some circumstances (e.g., when dealing with large windows), the second type of adaptation dynamically searches for the best unrolling factor for the current window size.

### 6.2.1 Runtime specialization

The “Runtime Specialization” strategy, depicted in Figure 6.17, uses the runtime generation features of Kadabra to rewrite the FIR method based on the window, reusing the previously generated code if the window does not change. The strategy uses the CreateAdapter aspect from the Kadabra API to map a given code template (specializeToWindow) to the target method (fir) and asks for a method adaptation before fir is called, but only if the window size did not change. To control this, it uses a field to know which window size was used. The template used in this strategy specializes the fir method by fully unrolling the innermost loop of the method and replaces the window accesses by the corresponding constant values.

```aspectdef
SpecializeFIR
adapter = call CreateAdapter('fir', 'specializeToWindow');
select class.method.call{'fir'}.arg{name=='c'} end
apply $oldSize = $class.exec newField(Types.int, 'oldSize');
$call.insert before %
  if([$arg]).length != [$oldSize]){
    [[adapter.adapt($arg)]]
    [[$oldSize]] = [[$arg]].length;
  %}
end
```

Figure 6.17. A simple approach to dynamically generate a specialized version of FIR when the coefficients window changes.
6.2.2 Runtime Exploration of Unroll Factors

The “Runtime Exploration of Unroll Factors” strategy requires more effort than the “Runtime Specialization” strategy since now one needs a way of trying versions of the \texttt{fir} method generated with different loop unrolling factors and keep the best version. Figure 6.18 partially depicts a strategy for that purpose. After using the previous strategy code, but with a different code template (\texttt{unrollByFactor}), it adds fields to store the exploration data: timer, current factor, best factor, best time, number of executions, etc. Then, the strategy has to insert code that controls the exploration. Before calling \texttt{fir}, the strategy observes the number of executions of the current version and, if the limit is achieved, it updates the best factor and adapts \texttt{fir} method with a new unroll factor. The execution time of \texttt{fir} is measured and the exploration finishes when the last factor, which would be a full unroll, is sampled and from that point on the version with the best unroll factor is used.

```aspectdef
UnrollFIR
input numRuns = 10 end
adapter = call CreateAdapter('fir', 'unrollByFactor');
select class.method.call('fir').arg{name=='c'} end
apply
timer = call NanoTimer();
$factor = $class.exec newField(mods, 'int', 'factor', 1);
$bestFactor: newField(mods, 'int', 'bestFactor', 0);
...
//more control variables
$call.insert before %{
  if(numRuns achieved){
    if(time improved){
      $bestFactor = newFactor;
    }
    newFactor = newFactor * 2;
    ... //reset counters and other logistic
  }
  [[timer.start()]]
}%;
$call.insert after %{
  [[timer.pause()]]
}%;
end end
```

Figure 6.18. Partial code of a strategy that explores \texttt{fir} versions with different unroll factors. The if conditions only contain a pseudo conditional expression for simplicity purposes.

6.2.3 Autotuner With Specialization

The previous strategies force the use of the specialized or an unrolled version of \texttt{FIR}. In scenarios in which these versions would perform worse than the original version, it would be necessary to roll back to the original version. An easy way of having this and a simpler way of writing this type of exploration would be the use of the autotuner API. This strategy makes use of the autotuner \texttt{Kadabra} API to have a way of comparing the specialized version with the original \texttt{fir} for different windows. The autotuner tries both versions and it decides which one is better to use for the given window. Figure 6.19 depicts this strategy. It extracts a functional interface based on the target method and builds an autotuner based on that new function interface. This autotuner will contain the default version of \texttt{fir} and a generative algorithm that uses the previous template
specializedToWindow') and the input window to generate the corresponding specialized version.

```javascript
aspectdef SpecializeFIR
select class.method.call{'fir'}.arg{name=='c'} end
apply
    fir = call ExtractFunctionalInterface($call);
    autotuner = new Autotuner(fir.$interface)
    .addAlgorithm(fir.default)
    .addGenerativeAlg('specializeToWindow', $arg)
    .addMonitor(Monitor.timer, 5, 10) //5 warmups, 10 executions
    .build(); //generate the autotuner code
end

autotuner.adapt(fir.$field);
autotuner.updateAndMeasure($call, $arg);
end
```

Figure 6.19. A strategy that builds an autotuner for the fir method that compares the original version with a specialized version generated with 'specializedToWindow' template.

### 6.2.4 Autotuner to Explore Unroll Factors

The “Autotuner to Explore Unroll Factors” strategy uses exactly the same strategy code as the previous one with the difference that the generative algorithm now uses an exploration configuration to try different unroll factors. Figure 6.20 shows the code to be added and changed for building the autotuner. It defines the range intended for the exploration and gives this range as the input for the generative algorithm. Since it does not specify the type of configuration then an exhaustive exploration is performed, i.e., versions are generated from value 2 to the size of the window ($arg) in increments of 2.

Comparing this strategy to “Runtime Exploration of Unroll Factors” strategy (see section 6.2.2), one can see that it is simpler to define the autotuner than the complete exploration code. Furthermore, Strategy 2 could not control different window sizes, which in turn the autotuner can do by specifying this window size as the key mapping to the current exploration.

```javascript
var factorRange = IntegerRange(2, $arg, 2);
autotuner = new Autotuner(fir.$interface)
    .addAlgorithm(fir.default)
    .addGenerativeAlg('specializeToWindow', factorRange).
...
```

Figure 6.20. A strategy that builds an autotuner for fir method that compares the original version with a specialized version generated with the 'specializedToWindow' template.

### 6.3 Median Smooth

In order to calculate the output value of each processed pixel, some image smoothing algorithms use the median of the N neighbors (where N = M*M, for an MxM window) of the corresponding input pixel, where N is defined prior to the median calculation. Figure 6.21 illustrates the smoothing operation of a pixel by using the median value of the surrounding neighbors. The operation uses a sliding window of size MxM (e.g., 3x3 in Figure 6.21) and, for each pixel, gets the neighbors in the window and calculates the median value by sorting the elements and retrieving the middle value. This median value is then assigned to the resulting output pixel.
The code implementation of a median image processing filter presented in (Fisher et al., 2005) is an example of code developed to be generic (from a library). The code considers the use of a java.util.ArrayList to store the neighbor values. Then, the algorithm determines the \( \lceil N/2 \rceil \) maximum values in the ArrayList and removes them. The last maximum value to be removed is the median value. This implementation may add performance overhead due to the selected data structure and there might be more efficient algorithms to calculate the median.

The following strategies were developed with the intent to find the best median calculator algorithm and to specialize that algorithm. The specialization process includes the selection of the best implementation for the given window size, and a runtime optimization phase, using the template-based generation that takes into account these values. For instance, the number of elements allows a fully unrolling of the innermost loop(s) iterating the neighbors. For the conducted experiments it is used the original version of the Hipr2 benchmark (Fisher et al., 2005), herein named as half-remove (HR), and the following three sorting algorithms:

- **CS**: Counting Sort
- **QS**: Quicksort
- **SN**: Sorting Network

Let us consider five different strategies to show the influence of having runtime adaptivity in the selection of algorithms and the importance of having code specialization associated with the adaptivity. The experiments with these strategies further demonstrate how LARA can be used to program autotuning strategies.

### 6.3.1 Extract Function Call

This first strategy is the first step when requiring a method selection/adaptation. It takes advantage of Java 8 functional programming (Warburton, 2014). Based on the target method, `medianNeighbor`, it generates a functional interface, named `IMedianNeighbor`, containing a method with the same signature as `medianNeighbor`. Then, using this new interface, it generates a new field in the class calling `medianNeighbor` and initializes it as default with `medianNeighbor`. The direct call to `medianNeighbor` is then replaced with an invocation over the newly added field.

The strategy definition is straightforward (see Figure 6.22) as the Kadabra API provides this functionality in an aspect named `ExtractFunctionalInterface`. 
Figure 6.22. A strategy that generates a new functional interface based on the signature of the method `medianNeighbor` and extracts calls to that method to a field with the type of the new functional interface.

### 6.3.2 Autotuner with Predefined Algorithms

After extracting the functional interface, one can now explore the algorithms available for calculating the median. The following strategy, called Autotuner, builds an autotuner with simple versions of the four algorithms. No code specialization or knobs are used in this strategy.

Since this strategy reuses the previous one, it only needs to specify the autotuning strategy inside the apply statement, as specified in Figure 6.23. The strategy starts by retrieving the algorithms to use (lines 2 to 5), builds an autotuner with these algorithms that uses execution time as the monitor (lines 7 to 13) and injects the adaptation code around the invocation to `medianNeighbor`. In line 7 one can see that it uses the `int` type so it can manage different explorations for the sizes the input argument can take. It then retrieves a specific exploration in line 15 based on the input size (second argument of `updateBefore`).

The autotuner uses the following logic to search/use the best version for the given kernel size. If it contains a version for the given size then it uses the mapped version, otherwise, it starts a new exploration in sampling mode. In this mode, each available version is measured during 10 executions, plus 5 more executions to consider as warmup (specified in line 12).

```plaintext
// Get sorting algorithms
$HR = GetMethod('HalfRemove', 'medianNeighbor');
$QS = GetMethod('Quicksort', 'getMedian');
$CS = GetMethod('CountingSort', 'getMedian');
$SN = GetMethod('SortingNetwork', 'getMedian');

//build an autotuner mapping explorations to specific #elements to sort
autotuner = new Autotuner(Types.int, median.$interface)
  .addAlgorithm($HR)
  .addAlgorithm($QS)
  .addAlgorithm($CS)
  .addAlgorithm($SN)
  .addMonitor(Monitor.timer, 5, 10); //5 warmups, 10 executions
.build(); //generate the autotuner code

autotuner.adapt(median.$field); // adapt based on the input
autotuner.updateBefore($call, $arg.length()); //length
autotuner.monitor($call); //length
```

Figure 6.23. Strategy to build an autotuner that manages explorations of sorting algorithms for the different sizes the input argument of `medianNeighbor` can take. The exploration uses execution time to evaluate the algorithms.

Figure 6.24 shows an example of an execution for an input of size 3. Every 15 iterations, a new version is provided until all version are explored, returning the best version (annotated with ‘*’). Here it is presented the execution time with (dark, dotted, blue) and without (light
blue) the adaptation overhead to demonstrate the difference between the execution times. In most
of the cases, the difference between these two is too small to be perceptible in the charts, as the
overhead is usually less than 1ms. In the first sampling, half-remove approach, it is observed the
influence of the overhead of the field access and the sampling-related statements.

![Graph](image)

Figure 6.24. Example of algorithm selection for a kernel of size 3. Each callout represents the start of a
version sampling. The callout with ‘*’ represents the point of best algorithm decision, from which all
executions with that kernel size use the selected version. The experiments were performed in a PC with
an Intel® Quad Core™i5 650 @ 3.20GHz CPU and 8 GB DDRAM.

6.3.3 Autotuner with Specialized Sorting Network and Counting Sort

The following strategy extends “Autotuner with Predefined Algorithms” strategy with a runtime
code specialization. Every time the number of elements changes, it replaces the (generic) sorting
network and counting sort with specialized versions of these algorithms, generated at runtime
with templates and based on the number of neighbors to sort (N). The template for sorting network
generates fully unrolled code with N local variables, initialized with a respective neighbor. The
method uses these local variables instead of array accesses. The counting sort template fully
unrolls the loops accessing the neighbors and replaces four local variables with the corresponding
constant value, calculated with the input size.

The strategy can be easily adjusted to include these templates by changing the lines 10 and
11 of Figure 6.23, as shown in the strategy in Figure 6.25, lines 5 and 6. One can replace the
invoked method with addAdaptiveAlgorithm and specify the template to be used for the
adaptation and the input the template requires, which is the number of elements.

```java
//build an autotuner mapping explorations to specific #elements to sort
autotuner = new Autotuner(Types.int, median.$interface)
    .addAlgorithm($HR)
    .addAlgorithm($QS)
    .addAdaptiveAlgorithm($CS, 'specializeCountingSort', $arg.length())
    .addAdaptiveAlgorithm($SN, 'specializeSortingNetwork', $arg.length())
    .addMonitor(Monitor.timer, 5, 10); //5 warmups, 10 executions
.build(); //generate the autotuner code
```

Figure 6.25. Strategy to build an autotuner that explores two normal sorting algorithms and two adaptive
algorithms.
6.3.4 Autotuner with Generative Algorithms

The “Autotuner with Generative Algorithms” strategy is similar to the “Autotuner with Specialized Sorting Network and Counting Sort” strategy, with a different approach for the specialization, as shown in Figure 6.26. Instead of replacing the original algorithms, every time the kernel size changes it generates, if necessary, a new class containing the specialized version. A version is reused if it was already generated for the current kernel size.

The main change in the previous strategy is that instead of using the `addAdaptiveAlgorithm` it now uses `addGenerativeAlgorithm`, which does not require the original versions as input and, thus, it also does not need to retrieve CS and SN at the beginning of the strategy.

```
autotuner = new Autotuner(Types.int, median.$interface)
...
.addGenerativeAlgorithm('specializeCountingSort', $arg.length())
.addGenerativeAlgorithm('specializeSortingNetwork', $arg.length())
...
```

Figure 6.26. Strategy to build an autotuner that explores two generative algorithms.

6.3.5 Switch to Pre-Selected Algorithms

The “Switch to Pre-Selected Algorithms” strategy would be used when considering an offline program profiling. After discerning the best version for each kernel size, a strategy can be defined to insert a switch in the code to shift to the best version and to generated specialized code if necessary. This approach uses the previous templates for sorting network and counting sort.

This aspect follows an “optimistic” approach where one knows the best versions at compile-time and define the strategy by statically mapping the best version to each kernel size by means of a switch statement. This switch considers three cases. In the first case, for kernel size 3, it generates a sorting network with that given size. The second and third case fall in the same statement, where a specialized counting sort is generated. As default case, it uses the original version. For the specialized versions, it uses the `ClassGenerator` aspect, available in Kadabra, providing the name of the template and the target method. Figure 6.27 shows this strategy. Lines 1 and 2 build two class generators, one for the sorting network and another for counting sort. Then, in lines 4 to 16, adaptivity code is inserted before the target loop, where the generators are used for specific kernel sizes and the original algorithm is used as the default method.

This strategy can be combined with any of the previous strategies. For instance, in the default case, one can use the autotuner to explore unknown sizes.
Case studies

```java
sortingNetGen = call ClassGenerator('specSortingNetwork', median.$interface);
countingSortGen = call ClassGenerator('specCountingSort', median.$interface);

$loop.insert before%
switch([[$arg]]){
    case 3:
        [[median.$field]] = [[sortingNetGen.generate($arg)]];
        break;
    case 5:
    case 7:
        [[median.$field]] = [[countingSortGen.generate($arg)]];
        break;
    default:
        [[median.$field]] = [[median.defaultMethod]];
}
}%
```

Figure 6.27. Strategy based on profiling information that dynamically selects, the best sorting algorithm and generates a specialized version, based on the number of elements to sort.

6.4 Summary

This chapter presented a set of case studies for which runtime adaptivity strategies were developed. The first case study addresses matrix multiplication. The strategies developed for this case study focused on dynamically search for the best block size for a set of loops to which the loop tiling transformation was applied. The second case study focuses on applying runtime code adaptation and specialization to a FIR method. The method is specialized to the input window via runtime adaptivity strategies and another strategy was also designed to explore and find the most suitable unroll factor for the innermost loop of that method. The third case addresses the exploration of multiple algorithms in a median smooth image processing operation. This exploration counts with different type of algorithms, from simple versions to generative algorithms.

All the case studies represent different runtime strategies and illustrate the LARA and Kadabra compatibility to help developers to extend existent applications or to develop new applications with runtime adaptivity features and schemes. In the following chapter, these strategies were evaluated not only in terms of improvements to the execution of the case studies but also in terms of productivity on the runtime adaptivity strategies definition.
Contents

7.1 Experimental Setup .................................................................................................................. 138
  7.1.1 Benchmarks and Strategies ............................................................................................... 138
  7.1.2 Execution Environment .................................................................................................... 139
7.2 Matrix Multiplication ............................................................................................................... 141
7.3 FIR .......................................................................................................................................... 144
7.4 Sorting Data ........................................................................................................................... 147
  7.4.1 Median Smooth ............................................................................................................... 147
  7.4.2 Integer Array Sort ......................................................................................................... 151
  7.4.3 Dataset Sorting ............................................................................................................. 153
7.5 Code Analysis and Metrics ..................................................................................................... 154
  7.5.1 Matrix Multiplication .................................................................................................... 154
  7.5.2 FIR ............................................................................................................................... 157
  7.5.3 Sorting Data .................................................................................................................. 159
7.6 Summary ............................................................................................................................... 161
Experimental Results

This chapter presents the experiments conducted to evaluate the approach proposed in this thesis. The opportunities for adaptation are analyzed in some kernels considering their possible use within a major application. In order to evaluate the feasibility of the proposed approach, a set of runtime adaptivity strategies are specified and applied to a set of application kernels on two target devices.

7.1 Experimental Setup

This section describes the setup used for the experiments and the metrics used to evaluate the proposed approach. The developed strategies of the case studies are used to generate different weaved versions of the applications that are then executed in different environments. A set of metrics are extracted from the source code of the applications, from the developed strategies and from execution profiling.

7.1.1 Benchmarks and Strategies

The evaluation considers three software tasks that can benefit from runtime adaptivity: matrix multiplication, a finite impulse response (FIR) filter and data sorting (see Chapter 6). For each of these tasks, it was specified strategies in LARA and the support proposed in this thesis to automatically apply those strategies was used. For the matrix multiplication task, it is proposed strategies that allow one to apply loop tiling and dynamically select an ideal tiling size according to the target system and execution environment. For the FIR task, it is proposed strategies for runtime specialization and dynamic exploration of loop unrolling factors. The sorting task is evaluated on three different use cases: a median smooth algorithm, sorting an array of integers and sorting an array of objects with respect to an integer field. These use cases count with several strategies that try to explore and select, at runtime, the best sorting algorithm for the given software and system context, including runtime generation of some of the algorithm implementations. Table 7-I summarizes the type of runtime adaptation applied by each strategy over the target benchmark. The table confirms that the experiments address all the adaptivity types discussed in the previous chapters, from software and compiler optimization knobs to the runtime algorithm selection and code generation.
Experimental Results

Table 7-I. Strategies and the corresponding types of runtime adaptation applied. “n.a.” is used when adaptivity is not applicable in that strategy, while “✓” depicts the use of an adaptivity type.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Strategy</th>
<th>Knobs</th>
<th>Compiler Optimizations</th>
<th>Multiple Algorithms</th>
<th>Multiple Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Multiplication</td>
<td>1-LoopInterchange</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td></td>
<td>2-FixedTile</td>
<td>n.a.</td>
<td>n.a.</td>
<td>✓</td>
<td>n.a.</td>
</tr>
<tr>
<td></td>
<td>3-Incremental</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4-Around</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5-AroundConc</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6-Random</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIR</td>
<td>1-Specialize</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2-UnrollExploration</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3-AutotunerSpec</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4-AutotunerUnroll</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MedianSmooth</td>
<td>1-FuncInterface</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td></td>
<td>2-TuneGeneric</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sort Integers</td>
<td>IntTuner</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sort Objects</td>
<td>ObjTuner</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The sections about the experiments are organized in terms of how a strategy developer may analyze and find possible adaptation opportunities and how he/she may incrementally develop adaptation strategies using LARA. These steps include the analysis over the target code, the compile-time changes that may need to be applied and the specification of different strategies, including aspects to be reused by multiple strategies.

7.1.2 Execution Environment

The experiments were executed in a PC with Ubuntu x64 14.04 LTS, Intel® Quad Core™i5 CPU 650 @ 3.20 GHz, 128Kb L1 cache and 512Kb L2 cache, with 8 GB DDRAM. The Java Virtual Machine used was the Java™ SE Runtime Environment (build 1.8.0_111), and the version 1.8.0.111 of javac was used for compiling the Java applications. Some of the experiments were also executed in a different environment, namely an Odroid system (HardKernel, 2013) with Ubuntu 12.04 Exynos 5422 Octa big.LITTLE ARM Cortex-A15 @ 2.0 GHz quad-core and Cortex-A7 quad-core CPUs, 8×32Kb L1 cache and 512Kb L2 cache, with 2 GB LPDDR3 RAM. This allows one to observe how different the optimizations applied behaves with the target system. These two systems are identified in the results as Desktop and Odroid, respectively.

7.1.3 Evaluation Criteria

The proposed strategies are evaluated using different phases. Firstly, it is presented the performance results of executing the different versions in different environments, including execution time improvements and the overheads of using the proposed strategies. These results show the execution improvements and indicate if runtime adaptivity was indeed useful.
Experimental Results

Then, an analysis is conducted to evaluate the LARA strategies do compensate the effort of writing them instead of manually changing the code based on the following parameters: how much effort is necessary to write a runtime strategy, how much effort would be necessary to change the code manually, how reusable are the aspects of other strategies and/or targets. For this evaluation, the analysis includes metrics from the LARA aspects applied and from both the original and the weaved code.

From the source code it is extracted information regarding application structure and the complexity of the code before (original version) and after the weaving process (weaved versions). Table 7-II shows the description of the extracted metrics. The number of logical LOCs and the number of types and members show how much the code increases after the weaving process. It is also presented metrics regarding the most advised method, namely the number of tokens, the cyclomatic complexity, the number of statements to show how complex the method becomes. The aspect ratio of that method is also measured to observe the percentage of code that was weaved in the method.

Table 7-II. Description of the metrics extracted from the Java source code.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>total logical LOCs of the application</td>
</tr>
<tr>
<td>Classes</td>
<td>number of classes</td>
</tr>
<tr>
<td>Interfaces</td>
<td>number of interfaces</td>
</tr>
<tr>
<td>Methods</td>
<td>number of methods</td>
</tr>
<tr>
<td>Fields</td>
<td>number of fields</td>
</tr>
<tr>
<td>Tokens</td>
<td>number of lexemes</td>
</tr>
<tr>
<td>CCN</td>
<td>cyclomatic complexity</td>
</tr>
<tr>
<td>Statements</td>
<td>number of statements in the program</td>
</tr>
<tr>
<td>Aspect Ratio</td>
<td>percentage of aspect statements weaved in the method</td>
</tr>
</tbody>
</table>

Regarding the LARA strategies, the evaluation considers information about the effort of writing LARA strategies vs the work required to make the changes over the source code by hand. This evaluation starts with the analysis of the written aspects strategies followed by metrics retrieved from the execution of the LARA strategies using the Kadabra Weaver.

The metrics of the written strategies are described in Table 7-III. and from the weaving process, it is extracted the metrics described in Table 7-IV.

Table 7-III. Description of the metrics extracted from the code of the strategies.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-LOC</td>
<td>total LOCs of the LARA aspects</td>
</tr>
<tr>
<td>Tokens</td>
<td>number of lexical tokens of the strategy</td>
</tr>
<tr>
<td>Aspects</td>
<td>number of aspects defined</td>
</tr>
<tr>
<td>API Calls</td>
<td>number of calls to the Kadabra API</td>
</tr>
<tr>
<td>Selects</td>
<td>number of selects done over the source code</td>
</tr>
<tr>
<td>Applies</td>
<td>number of applies written for the selected join points</td>
</tr>
<tr>
<td>Attributes</td>
<td>number of accessed attributes</td>
</tr>
<tr>
<td>Actions (inserts)</td>
<td>number of actions to be performed (from which the number of inserts)</td>
</tr>
<tr>
<td>N-LOC</td>
<td>number of native LOCs written to be weaved. Native code means literal code that is written in the native language (in this case Java) and it is directly weaved in the code. Code that is built by means of Node builders or actions is not included in this metric</td>
</tr>
<tr>
<td>Selected JPs</td>
<td>number of join points specified in select statements. E.g., select function. loop end counts as two selected join points</td>
</tr>
</tbody>
</table>
Experimental Results

Table 7-IV. Description of the metrics extracted from the weaving process.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aspects</td>
<td>executed aspects defined in the strategy</td>
</tr>
<tr>
<td>API Calls</td>
<td>executed aspects from the Kadabra API</td>
</tr>
<tr>
<td>Selects</td>
<td>selects performed</td>
</tr>
<tr>
<td>Applies</td>
<td>number of apply statements executed over the selected join points</td>
</tr>
<tr>
<td>Properties</td>
<td>number of accessed join point properties during the weaving process</td>
</tr>
<tr>
<td>actions (inserts)</td>
<td>actions performed over specific join points</td>
</tr>
<tr>
<td>N-LOC</td>
<td>total N-LOC weaved in the code</td>
</tr>
<tr>
<td>Selected JPs</td>
<td>total join points that had to be accessed while searching for the target join points. This also includes join points that do not completely match what he/she is selecting, but still, they have to be accessed to verify if they are what is wanted</td>
</tr>
<tr>
<td>Filtered JPs</td>
<td>the join points from Selected JPs that were filtered and considered for the select statements</td>
</tr>
<tr>
<td>Iterated JPs</td>
<td>join points that were iterated in apply statements, i.e., that passed the conditional statement. These include all consulted and advised join points</td>
</tr>
<tr>
<td>Advised JPs</td>
<td>join points in which an action was performed</td>
</tr>
<tr>
<td>Weaving Time (s)</td>
<td>time elapsed in seconds during the weaving process</td>
</tr>
<tr>
<td>CDLOC</td>
<td>concern diffusion over LOC (Eaddy et al., 2007), i.e., the number of transitions points between the original code and the aspect code added in the weaved code</td>
</tr>
<tr>
<td>Tangling ratio</td>
<td>the ratio between the number of transition points and the woven code LOC (Lopes and Kiczales, 1997)</td>
</tr>
<tr>
<td>Aspectual-Bloat</td>
<td>efficiency of aspect with respect to the woven code generated, i.e., the ratio between advised LOC and aspect LOC. A value of less than 1 means low efficiency of the aspect, while higher values mean high efficiency, i.e., the aspect produces more code than the size of the aspect (Lopes and Kiczales, 1997)</td>
</tr>
<tr>
<td>Similarity</td>
<td>code similarity between the original code and a weaved code</td>
</tr>
<tr>
<td>CAE</td>
<td>coupling on advice execution (Ceccato and Tonella, 2004), i.e., the number of aspects executed at the weaving phase</td>
</tr>
<tr>
<td>CDA</td>
<td>crosscutting degree of an aspect (Ceccato and Tonella, 2004), i.e., the number of modules (aka classes, interfaces and enums) advised by and aspect</td>
</tr>
<tr>
<td>CIM</td>
<td>coupling on intercepted modules (Ceccato and Tonella, 2004), i.e., the number of modules advised specifically selected in an aspect. Ideally, a reusable aspect has its value at zero.</td>
</tr>
</tbody>
</table>

7.2 Matrix Multiplication

The following experiment uses the matrix multiplication task presented in section 6.1. Table 7-V shows the performance of each strategy in charts, the total execution time and the total overhead of the runtime adaptation. Since strategies 1 and 2 do not contain any type of runtime adaptivity no overhead is considered.

Strategy 1 shows the performance of applying loop interchange and scalar replacement against the original version, providing a speedup of 8.67×, i.e., a decrease from 41.6 seconds to only 4.8 seconds per multiplication. However, it is clear from the other strategies that this is still not close to the optimum performance. Adding loop tiling with a fixed block of 256 (Strategy 2) improves nearly 1.3 seconds per multiplication. This value was adequate for the testing system, however, in different systems or runtime conditions it is possible that this fixed size may not be efficient. Strategies 3 to 6 try different block sizes, each with a different approach, and are able to find the best, or close to it. Strategy 3 is the one that requires more sampling to find the best value, since it executes seven configurations, and may execute more or less according to the matrices size. Strategy 6 executes less sampling versions but only because of the number of random configurations that were given, so it is only dependent on the number of configurations given in the strategy. Strategies 4 and 5 refer to the same search strategy, being the former sequential and the later concurrent to the application. These strategies are able to find a good
version with fewer iterations, always taking into account the L2 cache size of the current system processor. The main difference between these two strategies is the overhead of the concurrency, specifically, the thread start and stop operations and the use of atomic variables and channels.

Table 7-V. Strategies’ results for 2048×2048 shaped matrices, including the total execution time and the adaptation overhead.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Results for 2048×2048 matrices (100 executions)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1-LoopInterchange</strong></td>
<td>![Graph for Loop Interchange]</td>
</tr>
<tr>
<td>Loop Interchange</td>
<td></td>
</tr>
<tr>
<td>Scalar Replacement</td>
<td></td>
</tr>
<tr>
<td><strong>Time(s)</strong></td>
<td><strong>Overhead(ms)</strong></td>
</tr>
<tr>
<td>Original: 4 172</td>
<td>n.a.</td>
</tr>
<tr>
<td>LoopInt: 481</td>
<td></td>
</tr>
<tr>
<td><strong>2-FixedTile</strong></td>
<td>![Graph for FixedTile]</td>
</tr>
<tr>
<td>Loop tiling with block of 256</td>
<td></td>
</tr>
<tr>
<td><strong>Time(s)</strong></td>
<td><strong>Overhead(ms)</strong></td>
</tr>
<tr>
<td>352</td>
<td>n.a.</td>
</tr>
<tr>
<td><strong>3-Incremental</strong></td>
<td>![Graph for Incremental]</td>
</tr>
<tr>
<td>Loop tiling with incremental block 32→matrix size</td>
<td></td>
</tr>
<tr>
<td><strong>Time(s)</strong></td>
<td><strong>Overhead(ms)</strong></td>
</tr>
<tr>
<td>366</td>
<td>0.66</td>
</tr>
<tr>
<td><strong>4-Around</strong></td>
<td>![Graph for Around]</td>
</tr>
<tr>
<td>Loop tiling from half L2 cache size, explore surrounding values</td>
<td></td>
</tr>
<tr>
<td><strong>Time(s)</strong></td>
<td><strong>Overhead(ms)</strong></td>
</tr>
<tr>
<td>313</td>
<td>0.66</td>
</tr>
<tr>
<td><strong>5-AroundConc</strong></td>
<td>![Graph for AroundConc]</td>
</tr>
<tr>
<td>Loop tiling from half L2 cache size, explore surrounding values, use concurrency</td>
<td></td>
</tr>
<tr>
<td><strong>Time(s)</strong></td>
<td><strong>Overhead(ms)</strong></td>
</tr>
<tr>
<td>317</td>
<td>6.25</td>
</tr>
<tr>
<td><strong>6-Random</strong></td>
<td>![Graph for Random]</td>
</tr>
<tr>
<td>Loop tiling with random blocks</td>
<td></td>
</tr>
<tr>
<td><strong>Time(s)</strong></td>
<td><strong>Overhead(ms)</strong></td>
</tr>
<tr>
<td>344</td>
<td>5.71</td>
</tr>
</tbody>
</table>

*The dots (●) in the graph represent the execution of the best version instead of the sampling version, i.e., the moment the strategy is analyzing the current configuration and defining a new version.
Table 7-VI shows the average of the execution time and speedup of each strategy, respectively. Since the interest is to research the benefits of runtime adaptivity strategies, it is compared the multiple strategies against Strategy 1, as this strategy alone provides a great speedup against the original version even without any adaptivity. Strategy 2 achieves a higher speedup than Strategy 3, however, the latter is able to dynamically search for the best version, while the former may not suffice in some execution environments. Strategies 4 and 5 allow achieving the best speedups as they find a better version more quickly.

Table 7-VI. Average execution time and speedup of matrix multiplication against Strategy 1-LoopInterchange when multiplying matrices of size 2048, with loop tiling over two loops and the same block size assigned to both tiles.

<table>
<thead>
<tr>
<th>Version</th>
<th>Average Time (ms)</th>
<th>Average Speedup</th>
<th>#Blocks Sampled</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-LoopInterchange</td>
<td>4 806</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2-FixedTile</td>
<td>3 524</td>
<td>1.36</td>
<td>-</td>
</tr>
<tr>
<td>3-Incremental</td>
<td>3 662</td>
<td>1.31</td>
<td>7</td>
</tr>
<tr>
<td>4-Around</td>
<td>3 134</td>
<td>1.53</td>
<td>5</td>
</tr>
<tr>
<td>5-AroundConc</td>
<td>3 173</td>
<td>1.51</td>
<td>5</td>
</tr>
<tr>
<td>6-Random</td>
<td>3 437</td>
<td>1.40</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 7-VI shows the results when applying strategies that control two loop tilings with the same block size assigned to both loops, hence having a small number of sampled blocks sizes. In order to better understand how the exploration configuration influences the search for the best version and the continuous execution time Table 7-VII presents different experiments in which it is tried a different number of loops to which apply loop tiling and two exploration configurations, namely exhaustive (3-Incremental strategy) and local minimum (4-Around strategy). It also shows the difference between always using the same block size for all the tiled loops and using different blocks sizes for each loop.

It is seen that the best tuple of block sizes that give the best speedup is achieved when using the exhaustive approach, i.e., exploring all values, and using different blocks for each loop. However, this comes with the cost of longer execution time since all possible combinations must be sampled (81 for two loops, 729 for three loops). Using a local minimum strategy, the space exploration is reduced by 33% for two loops and almost 90% for three loops and still achieve a speedup very close to the best blocks tuples. By using the same block size for all loops one always have at most 9 combinations to sample for both two and three tiled loops, so the space exploration is small. Although this approach does not achieve the best combination, it still provides significant speedups and is a feasible approach if the exploration time is an issue.
Experimental Results

Table 7-VII. Results of executing the matrix multiplication with Strategies 1,3 and 4 over matrices of size 2048 by 2048, describing the execution time, the speedup against the version without loop tiling, and the number of explored version required to find the best version. The range given for the blocks is between 8 and 2048.

<table>
<thead>
<tr>
<th>Tiled Loops</th>
<th>Strategy</th>
<th>Same Block</th>
<th>Versions</th>
<th>Best Blocks</th>
<th>Best Time</th>
<th>Speedup</th>
<th>Avg. Time (ms)</th>
<th>Total Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1-LoopInterchange</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4.806</td>
<td>-</td>
<td>4.806</td>
<td>480.60</td>
</tr>
<tr>
<td>2</td>
<td>3-Incremental</td>
<td>Yes</td>
<td>9</td>
<td>(256, 256)</td>
<td>2.745</td>
<td>1.751</td>
<td>2.986</td>
<td>298.64</td>
</tr>
<tr>
<td></td>
<td>3-Incremental</td>
<td>No</td>
<td>81</td>
<td>(256, 2048)</td>
<td><strong>2.263</strong></td>
<td><strong>2.124</strong></td>
<td>4.836</td>
<td>483.57</td>
</tr>
<tr>
<td></td>
<td>4-Around</td>
<td>Yes</td>
<td>3</td>
<td>(128, 128)</td>
<td>3.225</td>
<td>1.490</td>
<td>3.229</td>
<td>322.94</td>
</tr>
<tr>
<td></td>
<td>4-Around</td>
<td>No</td>
<td>27</td>
<td>(16, 2048)</td>
<td>2.349</td>
<td>2.046</td>
<td><strong>2.488</strong></td>
<td><strong>248.78</strong></td>
</tr>
<tr>
<td>3</td>
<td>3-Incremental</td>
<td>Yes</td>
<td>9</td>
<td>(16,16,16)</td>
<td>2.325</td>
<td>2.067</td>
<td>2.349</td>
<td>2348.92</td>
</tr>
<tr>
<td></td>
<td>3-Incremental</td>
<td>No</td>
<td>729</td>
<td>(128,16,16)</td>
<td><strong>2.178</strong></td>
<td><strong>2.207</strong></td>
<td>4.456</td>
<td>4455.63</td>
</tr>
<tr>
<td></td>
<td>4-Around</td>
<td>No</td>
<td>76</td>
<td>(256,16,16)</td>
<td>2.18</td>
<td>2.205</td>
<td><strong>2.300</strong></td>
<td><strong>2299.61</strong></td>
</tr>
</tbody>
</table>

Since the adaptivity strategies are simple, the overhead for each multiplication is expected to be low, even for the concurrent versions. Table 7-VIII shows the average and the geometric mean of the strategies overhead. Overall, the overhead is indeed very small, being the concurrent versions (Strategies 5 and 6) the ones with higher overhead. Nonetheless, on average it is less than 0.002% for each multiplication operation.

Table 7-VIII. Results of the overhead of each strategy, including the average, the minimum and the maximum time.

<table>
<thead>
<tr>
<th>Version</th>
<th>Average (ns)</th>
<th>Average (%)</th>
<th>Min(ns)</th>
<th>Max(ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-Incremental</td>
<td>6 557</td>
<td>0.0002%</td>
<td>1 204</td>
<td>159 813</td>
</tr>
<tr>
<td>4-Around</td>
<td>6 611</td>
<td>0.0002%</td>
<td>1 199</td>
<td>157 928</td>
</tr>
<tr>
<td>5-AroundConc</td>
<td>61 276</td>
<td>0.0019%</td>
<td>22 730</td>
<td>2 238 903</td>
</tr>
<tr>
<td>6-Random</td>
<td>56 015</td>
<td>0.0017%</td>
<td>21 937</td>
<td>1 633 532</td>
</tr>
</tbody>
</table>

7.3 FIR

This experiment uses the strategies described in section 6.2 to improve the FIR method by means of runtime specialization. Figure 7.1 shows the average speedups when running the autotuners for specialization and unroll exploration in two different environments: Desktop and Odroid. The average speedup includes the speedups of all the explored versions. The versions that present speedups below one, i.e, loss of performance, is because of the average counts with the execution of all versions, including the original version in which they (usually) fall back.

For the Desktop the strategy is able to achieve better speedups (up to almost 3 times better than the original version) but it loses some performance when dealing with small data to process, especially for 100 elements. As the data size increases the gains are improved. Observing the specialization for data sizes higher than 10K one can compare the improvements on each window size and observe that the gains are comparably reduced as the window size increases. The unroll exploration only has gained for data size more than 100K and improves from $1.1\times$ to $2\times$ the original version.
Compared to the Desktop results, the Odroid system shows higher performance from more than 100 data elements and better gains for window size higher than 8, increasing performance from $2.2\times$ to $3\times$. Hence, the autotuning, and specially specialization, has better benefits in the Odroid system. The unroll exploration has benefited from 1K for most of the window kernel sizes and for more than 100K it achieves improvements between $1.5\times$ and $1.9\times$.

The autotuner version selection for each experiment is depicted in Table 7-IX. The decision of the best version varies between the systems, especially for data of 1K and 10K and for a window size of 32 for the loop unroll exploration strategy. For that window size, the Desktop version improves better if an unroll factor of 6 is used while for Odroid it is better to fully unroll the loop. For the other versions considering loop unrolling a full loop unroll was selected. Most of the selected versions were the specialized/unrolled versions.
## Experimental Results

Table 7-IX. The algorithm selected by the autotuner for each kernel size, between the original FIR (O), a specialized, generated, FIR (S), or an unrolled version with a specific factor (U [<factor>]).

<table>
<thead>
<tr>
<th>Platform</th>
<th>Data Size</th>
<th>Autotuner - Specialization</th>
<th>Autotuner - Unroll Exploration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>PC</td>
<td>100</td>
<td>S</td>
<td>O</td>
</tr>
<tr>
<td>Odroid</td>
<td>100</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

Table 7-X shows the average overhead added by the autotuning process per execution of fir, which includes measurements and the selection/generation of new versions. The overhead is always lower than 1%. The overhead is higher when dealing with small data sizes and for small window sizes. When dealing with big data sizes the overhead is so small compared to the execution time that is almost not felt during the execution.

Table 7-X. Average overhead (%) of the autotuning strategies, for each kernel size. The greener the cell is, the lower the overhead was.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Data Size</th>
<th>Autotuner - Specialization</th>
<th>Autotuner - Unroll Exploration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>PC</td>
<td>100</td>
<td>0.636</td>
<td>0.317</td>
</tr>
<tr>
<td></td>
<td>1K</td>
<td>0.687</td>
<td>0.092</td>
</tr>
<tr>
<td></td>
<td>10K</td>
<td>0.418</td>
<td>0.025</td>
</tr>
<tr>
<td></td>
<td>100K</td>
<td>0.078</td>
<td>0.009</td>
</tr>
<tr>
<td></td>
<td>1M</td>
<td>0.011</td>
<td>0.002</td>
</tr>
<tr>
<td>Odroid</td>
<td>100</td>
<td>0.001</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>1K</td>
<td>0.835</td>
<td>0.584</td>
</tr>
<tr>
<td></td>
<td>10K</td>
<td>0.765</td>
<td>0.352</td>
</tr>
<tr>
<td></td>
<td>100K</td>
<td>0.333</td>
<td>0.065</td>
</tr>
<tr>
<td></td>
<td>1M</td>
<td>0.056</td>
<td>0.008</td>
</tr>
<tr>
<td></td>
<td>10M</td>
<td>0.001</td>
<td>0.000</td>
</tr>
</tbody>
</table>
7.4 Sorting Data

The selection of the best sorting algorithm is a well-known area of research (Menon, 1986, Pasetto and Akhriev, 2011). An approach allowing multiple versions, each performing better for different criteria, such as the number of elements to sort and their range values, is in practice very useful. An adaptive application can select between different sorting algorithms, to measure their performance and keep track of best performing versions or even to explore the available versions when their performance is unknown for the given environment.

In the following experiments, it is shown to use the proposed approach to implement runtime adaptivity versions of common programs using sorting operations. It is considered here three use cases of sorting algorithms: an image processing algorithm using median, integer array sorting and a dataset being sorted by a specific attribute.

7.4.1 Median Smooth

These approaches were evaluated in an application executing the median smooth algorithm that changes the kernel size every 100 executions and all the experiments follow the same pattern of kernel sizes: 3 → 5 → 3 → 7 → 3 → 7 → 5. Table 7-VI compares the execution of the original version of MedianSmooth against the multiple strategies.

Each chart represents a program execution, using MedianSmooth, where the kernel is periodically changed to a different size. The markers in the horizontal axis represent the assignment of a new kernel size during execution, which is then executed for 100 iterations. The callouts containing an asterisk (*) represent the end of the sampling phase and the selection of the best algorithm, while the others represent access to the map to retrieve the best version for the current kernel size.

In 1-FuncInterface, the multiple executions of MedianNeighbor report similar values compared to the original version, with many fluctuations between the original and the field access of the strategy (sometimes the original is faster than 1-FuncInterface and vice-versa). The results in this chart represent the mean execution time of running 30 times both the original and the adapted version, reporting a slight improvement when using the field access. However, it is important to stress that it is not clear if this strategy actually optimizes the program execution. Nevertheless, this strategy is simply an assembling step, where the possible overhead of this strategy is expected to be overshadowed by the following strategies.

Compared to the original version, 2-TuneGeneric has an overall lower execution time even during the sampling phase. When the best version is already mapped, it is possible to observe a higher execution time reduction. Quicksort is the best approach for size 3, while counting sort provides the best execution time for kernels 5 and 7.
Table 7-XI. Results of executing MedianSmooth with the defined strategies. Total time of Original version: 10 144s.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Execution Timeline (in milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1-FuncInterface</strong></td>
<td><img src="image1" alt="Graph" /></td>
</tr>
<tr>
<td>• Extract functional interface</td>
<td></td>
</tr>
<tr>
<td>• Replace calls to medianNeighbor with new field with the type of new functional interface</td>
<td></td>
</tr>
<tr>
<td><strong>Time(s)</strong></td>
<td><strong>Overhead(ms)</strong></td>
</tr>
<tr>
<td>10 039</td>
<td>n.a.</td>
</tr>
</tbody>
</table>

| **2-TuneGeneric**      | ![Graph](image2)                      |
| • use 1-FuncInterface  |
| • autotuner with original algorithms |
| **Time(s)** | **Overhead(ms)** |
| 2 697        | 6.125                                |

| **3-TuneAdaptedAlg**   | ![Graph](image3)                      |
| • use 1-FuncInterface  |
| • autotuner with       |
| ◦ original HR and QS   |
| ◦ adapted CS and SN    |
| **Time(s)** | **Overhead(ms)** |
| 2748        | 139.480                              |

| **4-TuneGenAlg**       | ![Graph](image4)                      |
| • use 1-FuncInterface  |
| • autotuner with       |
| ◦ original HR and QS   |
| ◦ generative CS and SN |
| **Time(s)** | **Overhead(ms)** |
| 2755        | 55.387                                |

| **5-Switch**           | ![Graph](image5)                      |
| • Use 1-FuncInterface  |
| • Directly switch to the best version |
| • Best version considered at compile-time |
| • Generative CS and SN |
| **Time(s)** | **Overhead(ms)** |
| 822         | 27.19                                |
Experimental Results

In terms of performance, the improvements in 4-TuneGenAlg are similar to 3-TuneAdaptedAlg. The main difference is in the overhead. While strategy 3 requires regeneration for ad-hoc versions, this approach allows the maintenance of previously generated versions.

Strategy 5 (5-Switch) provides the best improvement of all strategies, as the selection of the best versions, for the three kernel sizes used, was defined prior to the application execution and at runtime, only a switch is required. The overhead, in this case, includes the runtime generation of sorting network and counting sort.

Table 7-XII depicts the average execution time of MedianSmooth for each strategy, where Sampling represents the average time of sampling all versions and Best is the execution of only the best, mapped, version.

With Strategy 2, during the sampling phase, the strategy is still able to provide an important speedup. When the best version is known, the program achieves speedups of $2\times$, $7\times$ and $13\times$. For kernel 7 it is seen an increase of more than $5\times$ from the sampling phase to the mapped version.

Strategy 3 brings favorable improvements in the execution. Compared to Strategy 2, not only the best version for kernel size 3 has changed to the specialized sorting network, it achieved more than $3\times$ speedup. The counting sort specialization further improved the execution time for kernels 5 and 7. This strategy, as well as 4, loses to strategy 2, in the sampling phase, for kernel size 7 due to the specialized version of counting sort being slower compared to the original counting sort (a slowdown of $0.73\times$).

One point of mention is the use of counting sort in these experiments. Here it is used a grayscaled image, in which the pixels can range between 0 and 255. Counting sort uses a table which size is the length of the input values range, in this case dealing with a table of size 256. In situations where the value range is higher, counting sort may not be feasible/suitable. Hence, whenever counting sort may not be used, or it is not available, the best approach for kernel size of 5 is the specialized sorting network, while for kernel 7 is the quicksort algorithm.

The Strategy 4 results are similar to 3, as the main difference between these two does not reside in the execution time, but in the overhead of each approach, the adaptation requirements and the system/program limitations. Both Strategies 3 and 4 use the same template for code generation. However, when a version is loaded from the map, and since Strategy 3 does not allow to have multiple versions generated at the same time, it might be needed to regenerate that version. One has to verify if the mapped version is a sorting network or a counting sort and if the version was specialized for that size. If it is for a different size, the method needs to be regenerated again. By using Strategy 4 one is able to generate ad-hoc classes containing a specialized method. This allows having multiple versions at the same time.
Table 7-XII. Average MedianSmooth execution time and speedup of the developed strategies.

<table>
<thead>
<tr>
<th>Version</th>
<th>Kernel Size</th>
<th>Sampling (ms)</th>
<th>Sampling Speedup</th>
<th>Best Version</th>
<th>Best (ms)</th>
<th>Best Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>289.50</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>Half-Remove</td>
<td>1301.04</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3337.41</td>
<td>1.00</td>
</tr>
<tr>
<td>1- FuncInterface</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>282.88</td>
<td>1.02</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>Half-Remove</td>
<td>1282.74</td>
<td>1.01</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3312.32</td>
<td>1.01</td>
</tr>
<tr>
<td>2- TuneGeneric</td>
<td>3</td>
<td>162.86</td>
<td>1.78</td>
<td>QuickSort</td>
<td>130.32</td>
<td>2.22</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>519.80</td>
<td>2.50</td>
<td>Counting Sort</td>
<td>187.50</td>
<td>6.94</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>1316.26</td>
<td>2.54</td>
<td>Counting Sort</td>
<td>250.13</td>
<td>13.34</td>
</tr>
<tr>
<td>3- TuneAdaptedAlg</td>
<td>3</td>
<td>108.98</td>
<td>2.66</td>
<td>Sorting Network</td>
<td>40.90</td>
<td>7.08</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>404.74</td>
<td>3.21</td>
<td>Counting Sort</td>
<td>154.50</td>
<td>8.42</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>1804.01</td>
<td>1.85</td>
<td>Counting Sort</td>
<td>194.31</td>
<td>17.18</td>
</tr>
<tr>
<td>4- TuneGenAlg</td>
<td>3</td>
<td>108.47</td>
<td>2.67</td>
<td>Sorting Network</td>
<td>39.90</td>
<td>7.26</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>406.74</td>
<td>3.20</td>
<td>Counting Sort</td>
<td>153.11</td>
<td>8.50</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>1810.82</td>
<td>1.84</td>
<td>Counting Sort</td>
<td>196.52</td>
<td>16.98</td>
</tr>
<tr>
<td>5- Switch</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>Sorting Network</td>
<td>39.67</td>
<td>7.30</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>Counting Sort</td>
<td>153.40</td>
<td>8.48</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>Counting Sort</td>
<td>197.37</td>
<td>16.91</td>
</tr>
</tbody>
</table>

Table 7-XIII presents the average overhead of the adaptation phase, in the execution of the smooth method, for each of the defined strategies. The smallest overhead is observed in Strategy 2. In this strategy, the adaptation consists of simply accessing the map or to execute in sampling mode. In sampling mode, one has first the overhead of one of the following: get current configuration, switch to the next configuration or finish sampling mode. In addition, while in sampling mode, after 10 (+5 for warmup) calls, the sampled version is compared with the current best before moving to the next configuration. Then it measures the time of the execution and afterward the update of the current sampling time. In average, the overhead is less than 0.005% in each medianNeighbor invocation (plus the field access overhead, previously mentioned).

The second strategy with the lowest overhead is Strategy 5, which consists of switching between versions, and generate if necessary, only when the kernel size changes. The code generation reflects in a maximum overhead of almost 19 ms, which is the first time code generation is invoked. Strategies 3 and 4 have the same overhead as Strategy 2 with extra overhead for the code generation and class (re)definition.
Experimental Results

Table 7-XIII. Average adaptation overhead, in microseconds, of each strategy for each execution of method smooth.

<table>
<thead>
<tr>
<th>Version</th>
<th>Average (μs)</th>
<th>Average (%)</th>
<th>Min(μs)</th>
<th>Max(μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strategy 2 (Select)</td>
<td>8.75</td>
<td>0.005</td>
<td>4.12</td>
<td>728.07</td>
</tr>
<tr>
<td>Strategy 3 (Specialize)</td>
<td>199.26</td>
<td>0.095</td>
<td>3.51</td>
<td>33 785.82</td>
</tr>
<tr>
<td>Strategy 4 (Generate)</td>
<td>79.13</td>
<td>0.021</td>
<td>3.67</td>
<td>26 338.75</td>
</tr>
<tr>
<td>Strategy 5* (Switch)</td>
<td>38.97</td>
<td>0.051</td>
<td>0</td>
<td>18 566.97</td>
</tr>
</tbody>
</table>

* The overhead is only when kernel size changes, i.e., during the iterations the overhead is zero.

Figure 7.2 shows the overhead of Strategies 3 and 4 during the execution. The overhead peaks represent a specialization request. Strategy 3, *Specialize*, always have a specialization request every time kernel size changes, while Strategy 4, *GenerateSpec*, is only invoked three times, i.e., every time a new value of kernel size occurs. Apart from the peaks, which depicts the algorithm generation/specialization, the overhead is usually less than 10ms. In exploration mode, as expected, there is a higher overhead since it is executed more adaptation-related statements, such as switch and measure versions.

![Figure 7.2. Overhead of specializing the sorting network vs generating a new, specialized, class.](image)

7.4.2 Integer Array Sort

When programming in Java, a very common approach to sort an array of integers is to use the method `java.util.Arrays.sort` provided by the Java API, which is a dual-pivot quicksort algorithm, an algorithm performing reasonably for most sorting scenarios. However, in a scenario where performance is essential, a selection of the best sorting algorithm is important. The best algorithm may not be easily found especially when the application executes in different environments with different resources.

Let us consider an autotuning strategy programmed in LARA, very similar to the ones specified in section 7.4.1, to explore three algorithms: quicksort (QS), counting sort (CS) and a parallel version of counting sort (CS#). In this parallel version of CS, each thread sorts a chunk of data using CS and in a final step, CS arrays (one per thread) are merged. This parallel CS version contains a knob for which it is also performed an exploration of the number of threads to use.

Figure 7.3 depicts experiments conducted with arrays with random values and considering different array sizes. The same experiment was conducted following three strategies: use
Experimental Results

quicksort, use counting sort, use autotuning to select between the two sorting algorithms. The two first strategies are used as a reference for the exploration to compare the scenarios when only one of the algorithms is used and when a strategy searches for a more adequate algorithm.

The search of the number of threads is performed with a steepest descent search around a starting value (here it is used the number of cores in the target architecture).

Figure 7.3. Sorting algorithm exploration to sort an array of integers (range=[0, 255]), using three versions: quicksort (QS), counting sort (CS) and parallel counting sort (CS# followed by the number of threads). Parallel counting sort uses a local minimum search starting with 4 threads. The vertical dotted lines mark the locations where the explored algorithm is switched and the best algorithm found is marked within a bubble. The array was initialized with random values before each sorting iteration.

As expected, the best algorithm varies according to the number of elements to sort. Although there are other properties that influence the performance of sorting algorithms, such as the type of the elements, the values range and the level of pre-sorting of the array, the experiments focus on the variation of the best algorithm according to the number of elements. The performance of quicksort reduces significantly as the number of elements to process increases, while counting sort maintains a steady performance. The best version selected for arrays with 10 elements was quicksort in the Desktop and counting sort in Odroid. The execution benefits from the exploration
Experimental Results

with speedups of $1.04 \times$ and $1.52 \times$ for 10 thousand elements and $6.65 \times$ and $5.75 \times$ for 10 million elements, for Desktop and Odroid, respectively, but with loss of performance, for arrays with 10 elements, due to the high cost of thread creation for the reduced number of elements to sort. The Odroid took more advantage of the exploration as the 10K version achieved a higher speedup than the Desktop and, besides Desktop having a higher speedup for 10M, the impact is more perceivable for Odroid as the execution time is reduced from 372 s to only 64 s.

7.4.3 Dataset Sorting

In this experiment, the sorting algorithms are used to sort, in ascending order, a “Census Income” dataset (Lichman, 2013) based on the age of a person. Loading the dataset results in a list of instances of “Person”, containing fields such as age, name and occupation. Since this use case requires ordering objects instead of arrays with primitive data types, the sorting algorithms used in this experiment are different. Specifically, the use case uses java.util.Collections.sort from Java and a CountingSort designed for a list of objects that accepts as input a function that extracts the value for sorting. A parallel version of CountingSort is also explored with the same search pattern as in the previous experiment. This experiment shows the reusability of strategies where only the algorithms to explore were changed.

Figure 7.4 shows the execution of the three algorithms. These executions consist of sorting 200 lists, with approximately 32,500 elements each, by the age of the Person. In the Desktop, Collections required 1067 ms to sort all the lists, while CountingSort required 337 ms (is less than a third of the time) and Exploration took 673 ms. In the Odroid Collections took 2908 ms, CountingSort 776 ms and Exploration 1277 ms. Similar to the previous results, the CountingSort was the fastest algorithm, with the exploration improving execution time of the application by $1.58 \times$ over Collections.
Experimental Results

Figure 7.4. Sorting algorithm exploration using the same exploration as in Figure 7.3 and considering the sorting of a collection of Person, ordered by age.

7.5 Code Analysis and Metrics

Here it is discussed the effort of writing the LARA strategies for runtime adaptivity. The discussion includes the complexity of the resulting weaved applications versus the required aspectual code necessary to produce those versions.

7.5.1 Matrix Multiplication

Table 7-XIV shows information about the original application and the weaved version generated by each strategy, and metrics about the two most advised methods: mult, the function multiplying two matrices, and callMult, a function that after some computation invokes function mult. The versions with higher complexity are the ones with exploration, adding more modules, members and statements within the code. The mult method increases its cyclomatic complexity after loop tiling is applied and callMult increases the number of statements by 56%, meaning that it contains substantial weaved code within it.
Experimental Results

Table 7-XIV - Java code before and after applying the different strategies

<table>
<thead>
<tr>
<th>Strategy</th>
<th>LOC</th>
<th>Classes</th>
<th>Interfaces</th>
<th>Methods</th>
<th>Fields</th>
<th>Most Affected Method</th>
<th>Tokens</th>
<th>CCN</th>
<th>Stmts</th>
<th>Aspect Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>193</td>
<td>4</td>
<td>0</td>
<td>18</td>
<td>3</td>
<td>callMult</td>
<td>60</td>
<td>2</td>
<td>4</td>
<td>n.a.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mult</td>
<td>140</td>
<td>4</td>
<td>9</td>
<td>n.a.</td>
</tr>
<tr>
<td>1-LoopInterchange</td>
<td>194</td>
<td>4</td>
<td>0</td>
<td>18</td>
<td>3</td>
<td>callMult</td>
<td>60</td>
<td>2</td>
<td>4</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mult</td>
<td>143</td>
<td>4</td>
<td>10</td>
<td>10%</td>
</tr>
<tr>
<td>2-FixedTile</td>
<td>194</td>
<td>4</td>
<td>0</td>
<td>18</td>
<td>5</td>
<td>callMult</td>
<td>60</td>
<td>2</td>
<td>4</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mult</td>
<td>215</td>
<td>6</td>
<td>12</td>
<td>25%</td>
</tr>
<tr>
<td>3-Incremental</td>
<td>228</td>
<td>6</td>
<td>0</td>
<td>23</td>
<td>7</td>
<td>callMult</td>
<td>139</td>
<td>2</td>
<td>9</td>
<td>56%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mult</td>
<td>215</td>
<td>6</td>
<td>12</td>
<td>25%</td>
</tr>
<tr>
<td>4-Around</td>
<td>228</td>
<td>6</td>
<td>0</td>
<td>23</td>
<td>7</td>
<td>callMult</td>
<td>139</td>
<td>2</td>
<td>9</td>
<td>56%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mult</td>
<td>215</td>
<td>6</td>
<td>12</td>
<td>25%</td>
</tr>
<tr>
<td>5-AroundConc</td>
<td>228</td>
<td>6</td>
<td>0</td>
<td>23</td>
<td>7</td>
<td>callMult</td>
<td>139</td>
<td>2</td>
<td>9</td>
<td>56%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mult</td>
<td>215</td>
<td>6</td>
<td>12</td>
<td>25%</td>
</tr>
<tr>
<td>6-Random</td>
<td>228</td>
<td>6</td>
<td>0</td>
<td>23</td>
<td>7</td>
<td>callMult</td>
<td>139</td>
<td>2</td>
<td>9</td>
<td>56%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mult</td>
<td>215</td>
<td>6</td>
<td>12</td>
<td>25%</td>
</tr>
</tbody>
</table>

Table 7-XV shows the difference between the work of writing LARA strategies using the functionalities of the Kadabra API and the actual execution of the strategies using the Kadabra weaver Table 7-XVI. Table 7-XV shows that the written strategies are small and defined in single aspects to define the whole strategy. This is due to the reusable strategies that were designed, namely Strategies 1, 2 (which uses Strategy 1) and TileAutotuner (which uses Strategies 1 and 2). Strategies 3 to 6 all reuse the TileAutotuner strategy, making these strategies very small as they are all invoking the same strategy but with different parameters: tile limits, configuration for the exploration and concurrency.

Table 7-XV – Strategies wrote in LARA

<table>
<thead>
<tr>
<th>Strategy</th>
<th>A-LOC</th>
<th>Tokens</th>
<th>Aspects</th>
<th>API Calls</th>
<th>Selects</th>
<th>Applies</th>
<th>Attributes</th>
<th>Actions (inserts)</th>
<th>N-LOC</th>
<th>Selected JPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-LoopInterchange</td>
<td>8</td>
<td>94</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>2 (0)</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>2-FixedTile</td>
<td>10</td>
<td>75</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>1 (0)</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>3-Incremental*</td>
<td>4</td>
<td>19</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4-Around*</td>
<td>6</td>
<td>40</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5-AroundConc*</td>
<td>6</td>
<td>42</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6-Random*</td>
<td>6</td>
<td>42</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>*TileAutotuner</td>
<td>22</td>
<td>179</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Despite being small strategies, the Kadabra weaver has a lot of work to perform. Table 7-XVI shows this disproportion as all the metrics specified in Table 7-XV increases based on how big the target application is and how much work should be done over it. The first two strategies show that the work done is actually from the code written in the strategies since these strategies do not use any functionality of the Kadabra API and only use actions to achieve their purpose. Based on the API calls it is seen that strategies 3 to 6 rely on the work done by the TileAutotuner strategy, with many executed select and apply statements and actions performed.

The join points (JPs), the attributes and the actions metrics are indeed interesting as they show the work done while visiting and weaving the source code. For instance, while the
Experimental Results

LARA code considers 12 join points within two select statements in 1-LoopInterchange, the weaver had to look at 67 join points, from which only 18 were considered as the actual pointcuts of the select statements and from these only 11 where actually iterated in an apply statement.

The autotuning strategies show that a lot of work has to be done to include the autotuning features. 258 join points are visited while processing, from which only 37 are actually used and only 10 are advised with actions. The strategies perform 13 code transformations and 5 code injections while using more than 150 join point attributes to perform decisions.

By means of these specific actions and built-in APIs, it is avoided the writing of native code and leave all the work to the API functionalities. The API has inner functions that define Java code directly into the application AST as nodes, a more secure approach rather than using literal code.

Table 7-XVI – Strategies executed with the Kadabra weaver

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Aspects</th>
<th>API Calls</th>
<th>Selects</th>
<th>Applies</th>
<th>Attributes</th>
<th>Actions (inserts)</th>
<th>N-LOC</th>
<th>Selected JPs</th>
<th>Filtered JPs</th>
<th>Iterated JPs</th>
<th>Advised JPs</th>
<th>Weaving Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-Loop Interchange</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>44</td>
<td>2 (0)</td>
<td>67</td>
<td>18</td>
<td>11</td>
<td>2</td>
<td>1.37</td>
<td></td>
</tr>
<tr>
<td>2-FixedTile</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>86</td>
<td>4 (0)</td>
<td>0</td>
<td>129</td>
<td>31</td>
<td>19</td>
<td>3</td>
<td>2.73</td>
</tr>
<tr>
<td>3-Incremental</td>
<td>1</td>
<td>10</td>
<td>12</td>
<td>10</td>
<td>155</td>
<td>18 (5)</td>
<td>12</td>
<td>258</td>
<td>50</td>
<td>37</td>
<td>10</td>
<td>3.49</td>
</tr>
<tr>
<td>4-Around Conc</td>
<td>1</td>
<td>11</td>
<td>12</td>
<td>10</td>
<td>155</td>
<td>18 (5)</td>
<td>12</td>
<td>258</td>
<td>50</td>
<td>37</td>
<td>10</td>
<td>3.35</td>
</tr>
<tr>
<td>5-Around Conc</td>
<td>1</td>
<td>11</td>
<td>12</td>
<td>10</td>
<td>155</td>
<td>18 (5)</td>
<td>12</td>
<td>258</td>
<td>50</td>
<td>37</td>
<td>10</td>
<td>3.77</td>
</tr>
<tr>
<td>6-Random</td>
<td>1</td>
<td>11</td>
<td>12</td>
<td>10</td>
<td>155</td>
<td>18 (5)</td>
<td>12</td>
<td>258</td>
<td>50</td>
<td>37</td>
<td>10</td>
<td>3.00</td>
</tr>
</tbody>
</table>

Table 7-XVII includes AO metrics to compare the weaved versions with the original version in order to analyze the effort of changing the code by hand vs writing LARA strategies. These strategies target a specific method and the calls to that method. Despite being a hotspot, if the method is only called in few locations it is expected to have small tangling ratio, i.e., the weaved code is not that sparse in the application code and is, in fact, more concentrated in specific locations. The aspectual bloat, size of the code vs weaved code, is really low in the first two strategies, as the first one only changes the order of two loops and the second only adds the tiling loop. Strategies 3 to 6 show high values of aspectual bloat since not only they are injecting code in the target methods but also generating new functionalities in the application.

In CAE can be seen the necessary aspects executed to achieve the strategy goals, where Strategies 3 to 6 require dozens of aspects to include autotuning features in the application. CDA shows that only two classes were the target of weaving and none of these two were specifically named in the strategies (CIM), meaning that the strategy would be potentially reusable for applications containing matrix multiplication.

Looking at the similarity of the weaved versions compared to the original version, they are pretty close to the original version as the application behavior and structure is maintained, and only the injected code and transformations change the similarity between versions. For instance, since Strategy 1 only interchanges two loops, the two versions are pretty similar. However, with loop tiling and autotuning features weaved in the code, it becomes less similar.
Table 7-XVII – AO metrics about the changes made over the source code with the written strategies

<table>
<thead>
<tr>
<th>Strategy</th>
<th>CDLOC</th>
<th>Tangling Ratio (%)</th>
<th>Aspectual Bloat</th>
<th>Similarity (%)</th>
<th>CAE</th>
<th>CDA</th>
<th>CBM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-LoopInterchange</td>
<td>4</td>
<td>2.06</td>
<td>0.13</td>
<td>99.80</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2-FixedTile</td>
<td>6</td>
<td>3.09</td>
<td>0.10</td>
<td>95.10</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3-Incremental</td>
<td>12</td>
<td>5.26</td>
<td>1.94</td>
<td>86.10</td>
<td>11</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>4-Around</td>
<td>12</td>
<td>5.26</td>
<td>1.67</td>
<td>85.80</td>
<td>12</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>5-AroundConc</td>
<td>12</td>
<td>5.26</td>
<td>1.59</td>
<td>85.80</td>
<td>12</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>6-Random</td>
<td>12</td>
<td>5.26</td>
<td>1.59</td>
<td>85.80</td>
<td>12</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

7.5.2 FIR

The resulting Java code of the strategies for FIR, as expected, has more complexity than the original version. Table 7-XVIII shows the growth for the different strategies. 1-Specialize strategy slightly increases the code and complexity by adding the code generation request while 2-UnrollExploration adds much more code, and cyclomatic complexity, in the target method since all the exploration code is inside this method. The autotuner aspects add many modules and members and only a few weaved statements (22%) since these strategies generate all the necessary autotuning and runtime generation code in the multiple modules and members.

Table 7-XVIII - Java code before and after applying the different strategies

<table>
<thead>
<tr>
<th>Strategy</th>
<th>LOC</th>
<th>Classes</th>
<th>Interfaces</th>
<th>Methods</th>
<th>Fields</th>
<th>Most Affected Method</th>
<th>Tokens</th>
<th>CCN</th>
<th>Stmts</th>
<th>Aspect Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>120</td>
<td>2</td>
<td>0</td>
<td>12</td>
<td>1</td>
<td>dataFilter</td>
<td>207</td>
<td>6</td>
<td>18</td>
<td>0%</td>
</tr>
<tr>
<td>1-Specialize</td>
<td>140</td>
<td>3</td>
<td>0</td>
<td>14</td>
<td>4</td>
<td>dataFilter</td>
<td>246</td>
<td>7</td>
<td>21</td>
<td>14%</td>
</tr>
<tr>
<td>2-UnrollExploration</td>
<td>177</td>
<td>4</td>
<td>0</td>
<td>14</td>
<td>12</td>
<td>dataFilter</td>
<td>449</td>
<td>13</td>
<td>42</td>
<td>57%</td>
</tr>
<tr>
<td>3-AutotunerSpec</td>
<td>yu</td>
<td>5</td>
<td>1</td>
<td>20</td>
<td>5</td>
<td>dataFilter</td>
<td>287</td>
<td>6</td>
<td>23</td>
<td>22%</td>
</tr>
<tr>
<td>4-AutotunerUnroll</td>
<td>176</td>
<td>5</td>
<td>1</td>
<td>20</td>
<td>5</td>
<td>dataFilter</td>
<td>287</td>
<td>6</td>
<td>23</td>
<td>22%</td>
</tr>
</tbody>
</table>

Table 7-XIX shows the effort of writing the LARA strategies. Code specialization is achieved with only two actions: add a field to keep track of generated versions and inject code to request code generation. Unroll Exploration has a lot of work written, with almost three times A-LOC compared to the other strategies, since the entire exploration code written as a strategy. This includes the use of the runtime code specialization API, control fields and native code injection. The Spec and Unroll strategies focus on using the autotuning features of the Kadabra API and thus only have to take care of where and how the specialization is applied, leaving the application of actions to the API, thus having small-sized strategies that focus on selecting and analyzing join points for the autotuning.
Experimental Results

Table 7-XIX – Strategies wrote in LARA

<table>
<thead>
<tr>
<th>Strategy</th>
<th>A-LOC</th>
<th>Tokens</th>
<th>Aspects</th>
<th>API Calls</th>
<th>Selects</th>
<th>Applies</th>
<th>Attributes</th>
<th>Actions (inserts)</th>
<th>N-LOC</th>
<th>Selected JPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-Specialize</td>
<td>24</td>
<td>126</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>2 (1)</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>2-UnrollExploration</td>
<td>63</td>
<td>273</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>9 (3)</td>
<td>25</td>
<td>4</td>
</tr>
<tr>
<td>3-AutotunerSpec</td>
<td>27</td>
<td>208</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>4-AutotunerUnroll</td>
<td>27</td>
<td>226</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

Similar to the results of the matrix multiplication experiments, the execution of the LARA strategies shows, in Table 7-XX, a lot more work than the one written in the LARA aspects. Strategies 1-Specialize and 2-UnrollExploration had to verify a lot of join points and their attributes, in order to add runtime specialization, but only 19/15 were analyzed and only 4/5 where advised. The work done by these strategies focuses more on the selects and actions applied and the API was for the code specialization, measurements and control features. Again, although one needs to write smaller strategies when using the autotuning API, the work performed by the API and the weaver is much more substantial, as seen by the number of API calls, actions applied and the iterated join points. Also, by using the APIs one avoids writing and directly injecting native code.

Table 7-XX – Strategies executed with Kadabra weaver

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Aspects</th>
<th>API Calls</th>
<th>Selects</th>
<th>Applies</th>
<th>Attributes</th>
<th>Actions (inserts)</th>
<th>N-LOC</th>
<th>Selected JPs</th>
<th>Filtered JPs</th>
<th>Iterated JPs</th>
<th>Advised JPs</th>
<th>Weaving Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-Specialize</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>588</td>
<td>5 (2)</td>
<td>6</td>
<td>519</td>
<td>20</td>
<td>19</td>
<td>4</td>
<td>2.14</td>
</tr>
<tr>
<td>2-UnrollExploration</td>
<td>2</td>
<td>7</td>
<td>6</td>
<td>3</td>
<td>567</td>
<td>15 (4)</td>
<td>30</td>
<td>389</td>
<td>16</td>
<td>15</td>
<td>5</td>
<td>3.12</td>
</tr>
<tr>
<td>3-AutotunerSpec</td>
<td>2</td>
<td>12</td>
<td>9</td>
<td>10</td>
<td>121</td>
<td>20 (4)</td>
<td>11</td>
<td>135</td>
<td>28</td>
<td>25</td>
<td>8</td>
<td>4.58</td>
</tr>
<tr>
<td>4-AutotunerUnroll</td>
<td>2</td>
<td>12</td>
<td>9</td>
<td>10</td>
<td>123</td>
<td>20 (4)</td>
<td>11</td>
<td>135</td>
<td>28</td>
<td>25</td>
<td>8</td>
<td>4.91</td>
</tr>
</tbody>
</table>

Table 7-XXI depicts the AO metrics regarding the LARA strategies for runtime specialization of FIR. Once again, since the strategies are targeting a specific code to optimize, they do not tend to weave the code in different locations, thus having a small tangling ratio. The first two strategies show a low value of aspectual bloat since the size of the strategies are slightly bigger than the injected code12, while the autotuner strategies have high values of aspectual bloat. The relation between the Aspectual Bloat and the CAE allows concluding that these aspectual bloat values are due to the use of more API calls than by completely writing the strategies using only the “native” functionalities, ergo actions, of the weaving framework.

---

12 An important note here is that the code template is not being considered in the metrics calculations, as the way of defining the templates is an orthogonal problem. For instance, the templates used in “1-Specialization” strategy takes as much as 150 Java+Bytecode LOCs.
Experimental Results

Table 7-XXI – AO metrics about the changes made over the source code with the written strategies

<table>
<thead>
<tr>
<th>Strategy</th>
<th>CDLOC</th>
<th>Tangling Ratio (%)</th>
<th>Aspectual Bloat</th>
<th>Similarity (%)</th>
<th>CAE</th>
<th>CDA</th>
<th>CBM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-Specialize</td>
<td>6</td>
<td>4.29</td>
<td>0.83</td>
<td>91.60</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2-UnrollExploration</td>
<td>6</td>
<td>3.39</td>
<td>0.90</td>
<td>83.10</td>
<td>9</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3-AutotunerSpec</td>
<td>6</td>
<td>3.41</td>
<td>2.07</td>
<td>81.00</td>
<td>14</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4-AutotunerUnroll</td>
<td>6</td>
<td>3.41</td>
<td>2.07</td>
<td>80.40</td>
<td>14</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

7.5.3 Sorting Data

Table 7-XXII contains metrics about the generated code from the strategies applied to the three different sorting problems. The MedianSmooth task has its code increased from 13% and 23.5% for the autotuning features. The 3-TuneAdaptedAlg strategy is the most expensive one with 23.5% increase of code, introducing 4 classes, 11 methods and 9 fields. The target method has more than 45% of aspectual code weaved (the autotuning decisions code) while maintaining the inner CCN intact. The strategy 5-Switch introduces smaller code parcels since no exploration is needed and the autotuning is done by the “static” switch, however, having more code weaved and higher complexity in the target method.

As for the Integers and Objects sorting problem, it is observed a similar increment between these two use cases (around 12%), since the problem is very similar and the main difference is the use of Objects in one use case (which requires a comparator function) and a primitive type in another (direct comparison of values).

Table 7-XXII - Java code before and after applying the different strategies over MedianSmooth and Integers and Objects sorting use cases.

<table>
<thead>
<tr>
<th>Target</th>
<th>Strategy</th>
<th>LOC</th>
<th>Classes</th>
<th>Interfaces</th>
<th>Methods</th>
<th>Fields</th>
<th>Most Affected Method</th>
<th>Tokens</th>
<th>CCN</th>
<th>Stmts</th>
<th>Aspect Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>MedianSmooth</td>
<td>Original</td>
<td>339</td>
<td>6</td>
<td>26</td>
<td>5</td>
<td></td>
<td>smooth</td>
<td>111</td>
<td>3</td>
<td>6</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>1-FuncInterface</td>
<td>343</td>
<td>6</td>
<td>1</td>
<td>27</td>
<td>6</td>
<td>smooth</td>
<td>111</td>
<td>3</td>
<td>6</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>2-TuneGeneric</td>
<td>383</td>
<td>8</td>
<td>1</td>
<td>33</td>
<td>9</td>
<td>smooth</td>
<td>189</td>
<td>11</td>
<td>45%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3-TuneAdaptedAlg</td>
<td>419</td>
<td>10</td>
<td>1</td>
<td>37</td>
<td>14</td>
<td>smooth</td>
<td>193</td>
<td>12</td>
<td>50%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4-TuneGenAlg</td>
<td>403</td>
<td>9</td>
<td>1</td>
<td>35</td>
<td>10</td>
<td>smooth</td>
<td>193</td>
<td>12</td>
<td>50%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5-Switch</td>
<td>364</td>
<td>6</td>
<td>1</td>
<td>29</td>
<td>6</td>
<td>smooth</td>
<td>166</td>
<td>16</td>
<td>63%</td>
<td></td>
</tr>
<tr>
<td>Integers</td>
<td>Original</td>
<td>379</td>
<td>4</td>
<td>0</td>
<td>21</td>
<td>3</td>
<td>sorter</td>
<td>159</td>
<td>12</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IntTuner</td>
<td>425</td>
<td>6</td>
<td>1</td>
<td>28</td>
<td>7</td>
<td>sorter</td>
<td>231</td>
<td>17</td>
<td>29%</td>
<td></td>
</tr>
<tr>
<td>Objects</td>
<td>Original</td>
<td>372</td>
<td>4</td>
<td>0</td>
<td>22</td>
<td>4</td>
<td>sorter</td>
<td>146</td>
<td>4</td>
<td>6</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>ObjTuner</td>
<td>418</td>
<td>6</td>
<td>1</td>
<td>29</td>
<td>8</td>
<td>sorter</td>
<td>222</td>
<td>11</td>
<td>45%</td>
<td></td>
</tr>
</tbody>
</table>

Metrics about the written strategies for these problems are depicted in Table 7-XXIII. Since the autotuning strategies reuse the same base strategy, AbstractAutotuner, the strategies have small aspects that focus on listing the algorithms to use inside the autotuner while focusing on building the autotuning structure. Strategy 2-TuneGeneric focuses on selecting the generic sort algorithms available in the use case while Strategies 3 and 4 require more API calls to dynamically generate/adapt methods. Strategy 5-Switch uses more native code to include the switch for the algorithm and also requires API calls in order to add runtime code specialization. The IntTuner and ObjectTuner strategies are fair examples of complete aspect reusability since they both use the same LARA strategy and thus having the same values.
Experimental Results

Table 7-XXIII – Strategies were in LARA

<table>
<thead>
<tr>
<th>Strategy</th>
<th>A-LOC</th>
<th>Tokens</th>
<th>Aspects</th>
<th>API Calls</th>
<th>Selects</th>
<th>Applies</th>
<th>Attributes</th>
<th>Actions (inserts)</th>
<th>N-LOC</th>
<th>Selected JPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-FuncInterface</td>
<td>7</td>
<td>35</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2-TuneGeneric*</td>
<td>11</td>
<td>55</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3-TuneAdaptedAlg*</td>
<td>18</td>
<td>137</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4-TuneGenAlg*</td>
<td>15</td>
<td>110</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5-Switch</td>
<td>23</td>
<td>92</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>(1)</td>
<td>9</td>
</tr>
<tr>
<td>*AbstractAutotuner</td>
<td>27</td>
<td>208</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>8</td>
<td>2</td>
<td>(1)</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 7-XXIV depicts the execution of the LARA strategies. The two strategies with code specialization show more weaver work than the others in a substantial way. More than 600 join points and their attributes were consulted and from them, more than 40 were iterated, with more than 20 actions applied to add autotuning features with runtime specialization. Comparing to the results of the written strategies and looking at the number of API calls it can be seen how much of this work was hidden by the API functionalities. If one was to manually add these features in the application, it would have to consider all this cumbersome work, including the actions and changes to be applied. The results of 2-TuneGeneric, IntTuner and ObjTuner are close since these three strategies are very much similar, as they focus on simply using the existing algorithms.

Table 7-XXIV – Strategies executed with *Kadabra* weaver

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Aspects</th>
<th>API Calls</th>
<th>Selects</th>
<th>Applies</th>
<th>Attributes</th>
<th>Actions (inserts)</th>
<th>N-LOC</th>
<th>Selected JPs</th>
<th>Filtered JPs</th>
<th>Iterated JPs</th>
<th>Advised JPs</th>
<th>Weaving Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-FuncInterface</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>104</td>
<td>2</td>
<td>0</td>
<td>140</td>
<td>17</td>
<td>14</td>
<td>2</td>
<td>1.87</td>
</tr>
<tr>
<td>2-TuneGeneric</td>
<td>2</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>140</td>
<td>18</td>
<td>(4)</td>
<td>11</td>
<td>179</td>
<td>31</td>
<td>28</td>
<td>3.12</td>
</tr>
<tr>
<td>3-TuneAdaptedAlg</td>
<td>2</td>
<td>16</td>
<td>18</td>
<td>14</td>
<td>632</td>
<td>26</td>
<td>(5)</td>
<td>12</td>
<td>656</td>
<td>51</td>
<td>44</td>
<td>12</td>
</tr>
<tr>
<td>4-TuneGenAlg</td>
<td>2</td>
<td>17</td>
<td>17</td>
<td>14</td>
<td>592</td>
<td>23</td>
<td>(5)</td>
<td>12</td>
<td>655</td>
<td>48</td>
<td>41</td>
<td>9</td>
</tr>
<tr>
<td>5-Switch</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>293</td>
<td>5</td>
<td>(1)</td>
<td>13</td>
<td>345</td>
<td>26</td>
<td>23</td>
<td>3.23</td>
</tr>
<tr>
<td>IntTuner</td>
<td>2</td>
<td>10</td>
<td>11</td>
<td>10</td>
<td>82</td>
<td>18</td>
<td>(4)</td>
<td>11</td>
<td>158</td>
<td>30</td>
<td>27</td>
<td>7</td>
</tr>
<tr>
<td>ObjTuner</td>
<td>2</td>
<td>10</td>
<td>11</td>
<td>10</td>
<td>146</td>
<td>18</td>
<td>(4)</td>
<td>11</td>
<td>165</td>
<td>29</td>
<td>26</td>
<td>6</td>
</tr>
</tbody>
</table>

The applied strategies have once again a low tangling ratio (≈ 1%), as shown in Table 7-XXV, due to the fact of applying the strategies over a bottleneck focused on a specific part of the code. One important advantage here is the reusability of the *AbstractAutotuner* to try different exploration strategies, and this is shown in the aspectual bloat values that show more than 4 units while executing between 11 to 19 aspects.
Experimental Results

Table 7-XXV – AO metrics about the changes made over the source code with the written strategies

<table>
<thead>
<tr>
<th>Strategy</th>
<th>CDLOC</th>
<th>Tangling Ratio (%)</th>
<th>Aspectual Bloat</th>
<th>Similarity(%)</th>
<th>CAE</th>
<th>CDA</th>
<th>CBM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-FuncInterface</td>
<td>4</td>
<td>1.17</td>
<td>0.57</td>
<td>99.10</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2-TuneGeneric</td>
<td>6</td>
<td>1.57</td>
<td>4.00</td>
<td>92.80</td>
<td>11</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3-TuneAdaptedAlg</td>
<td>6</td>
<td>1.43</td>
<td>4.44</td>
<td>88.80</td>
<td>18</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4-TuneGenAlg</td>
<td>6</td>
<td>1.49</td>
<td>4.27</td>
<td>90.80</td>
<td>19</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5-Switch</td>
<td>6</td>
<td>1.65</td>
<td>1.09</td>
<td>95.60</td>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>IntTuner</td>
<td>6</td>
<td>1.41</td>
<td>1.39</td>
<td>93.30</td>
<td>12</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ObjTuner</td>
<td>6</td>
<td>1.61</td>
<td>1.39</td>
<td>93.30</td>
<td>12</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

7.6 Summary

This chapter presented the experimental results of applying runtime adaptivity strategies to a set of case studies and executing the application in two different environments. The case studies consider most dimensions of the runtime adaptivity and autotuning approach and thus contributes to identify practical aspects regarding the proposed approach and to test and evaluate all the approach, from the LARA and API use to the Kadabra source to source compiler. In addition, the programming strategies presented allow interesting performance improvements and can be used in deployment scenarios.

The experiments show that the specification of runtime adaptivity strategies, in general, allowed us to improve the performance of insignificant values. The loss of performance is observed in some situations, especially in cases where the execution time of the code to adapt was too short and the adaptivity strategies added unnecessary overhead. This enforces the idea that runtime adaptivity strategies must be carefully planned not only where to apply but in which conditions should be applied. For instance, only apply runtime adaptivity when FIR is processing more than 10K values. We note however that this kind of decisions can be included as LARA strategies.

Runtime specialization proved to be an important asset for algorithm improvement, taking advantage of applying compile-time optimizations at runtime with the template-based approach. The autotuning feature is, in fact, an important feature of the proposed approach that allows developers to easily define runtime adaptivity strategies for multiple purposes, including knob values exploration and algorithm selection. Furthermore, the possibility of defining the type of configuration one wants to use gives him/her more decision freedom. For instance, if one has to reduce the exploration space for time limitations or to avoid versions explosion, one can opt to use a local minimum approach.

The experiments also show that the proposed approach has low overhead and for heavy processes is almost negligible. Locations in which the overhead may be less negligible would be during code generation, version analysis and measurements.

The written runtime adaptivity strategies were shown by the metrics to be small not only compared to the weaved code but also to the necessary effort to achieve that same weaved code. The metrics showed that the strategies of these experiments had low aspectual bloat since they are aimed at specific code parcels and mainly because the target method for autotuning is called.
Experimental Results

in a specific place in the code and it is not sparse in the target applications. The aspectual bloat and CAE show that in most cases a lot of work is done by defining small strategies that use API functionalities. Many of the work done by the weaver is hidden by the use of the actions of **Kadabra** and the **Kadabra** API. Even the ones using more LARA code than the one injected are more useful than changing the code by hand as they avoid multiple versions of the same application and contribute to maintainability. As already shown, many of the transformations are less error-prone when applied in a semi-automated way and having a way of trying different strategies increases the strategies potential.
8

Conclusions

Contents

8.1 Concluding Remarks ................................................................. 164
8.2 Future Work ........................................................................... 166
Conclusions

This chapter presents the final remarks on the work of this thesis as well as suggestions for future work.

8.1 Concluding Remarks

This thesis focused on research addressing a new aspect-oriented programming (AOP) approach for adapting applications at runtime. It proposes novel techniques to program strategies for runtime adaptivity and methods to interface the application and program execution elements to the behavior specified by the code of the strategies. The work aimed at a flexible programming model and compiler techniques able to make easier the programming and evaluation of runtime adaptivity strategies. An approach promoting the separation of concerns makes the debug, verification, optimizations, specialization and the mapping of the behavior responsible for the runtime adaptation to specific computing cores easier. Mainly, the proposed approach was focused on imposing minor changes over the original application. This promotes the portability of the strategies, both in terms of target language and system and the exploration of different strategies in early development cycles. To this end, LARA, a DSL with aspect-oriented concepts, is used to allow the specification of user-defined adaptation strategies. The approach relies on using the DSL to program adaptation strategies decoupled from the target application.

The approach allows the design of runtime adaptivity strategies divided into two phases: compile-time changes and runtime adaptivity specification. At the compile-time phase, the source code is restructured with new features in order to allow adaptivity at runtime, exposing the availability of relevant information and the runtime access to program elements. The runtime adaptivity specification phase includes the adaptivity for the application. The approach focuses on both adaptivity at the application level and at the code generation level. At the application level, software and system parameters can be dynamically configured or an algorithm can be specifically selected. At the code generation level, it is possible to develop strategies that generate new versions of a method. The runtime generation allows to explore different versions of the same code, each with different optimizations and/or optimization parameter values, to try different implementations of an algorithm, and even to take advantage of the information of the execution to develop versions that specialize to that information. For the runtime generation, a low-level template-based code generator approach was proposed. This approach is based on templates containing low-level code segments (e.g. Java bytecodes) annotated with compiler optimizations that, based on input data provided at runtime, generate the actual low-level code to be executed. A low-level approach avoids the overhead of compiling high-level code at runtime.

The approach considers that the specification of runtime adaptivity has to deal with a set of essential points of interest that define where, when and how the adaptivity takes place. The strategy specification must define and prepare where the application is adapted, include when and how frequently the adaptation occurs and how is the adaptation code executed, e.g. parallel to the application execution.

The runtime adaptation is leveraged with an autotuning approach for the discussed adaptation levels. The autotuner allows the runtime selection of a configuration for scenarios in which the best configuration is known, as well as an exploration mode to explore different configurations for unknown/unexpected scenarios. The autotuning can be used for providing configurations for multiple target parameters, for providing algorithms to be executed or for
Conclusions

providing a combination of both levels, i.e., provide a configuration for some parameters when a specific algorithm is used. The algorithms include the use of precompiled versions and the runtime, template-based, method generation. The autotuner can take advantage of the template-based code generation to try different configurations for the generation, allowing the exploration of different versions. The exploration mode is supported with search schemes that allow strategies to reduce the exploration space and with the specification of the adaptation goals. A goal is specified as criteria that compare the scores of the different explored configurations and decides which version is considered the best. A configuration is given a score by continuously measuring the execution of that configuration while it is being sampled.

The approach provides high-levels of flexibility and is helpful to assist developers when considering the application and evaluation of online autotuning algorithms. The approach allows developers to easily evaluate different execution schemes for autotuning, e.g., the use of thread-based schemes where the calculations of the tuning values can be in parallel to the execution of the application, versus a scheme where those calculations are embedded in the application as sequential code, and for the sampling and production stages.

This thesis contributes with a weaving framework and a runtime adaptivity API developed to realize the proposed approach. A framework for the LARA language was developed not only to improve the available features of the LARA language but also to improve the development of new weaving environments, to ease the update and the extent of these environments and to allow more aspect decomposition for more reusability. This framework includes a generic version of an interpreter for the LARA language (larai) and a weaver generator that develops new weaving environments that interface with larai. This interface allows one to easily develop new weaver engines where the developer only focuses on the integration of the larai interface with a source-to-source compiler, leaving all the LARA interpretation and execution to larai. The framework also includes new mechanisms for improved aspect composition and for the definition of API-based strategies to allow more reusability.

The improvements in the LARA language and the development of the LARA framework allowed an easier integration of LARA with Object-Oriented programming languages and more specifically the development and integration of a Java-to-Java weaving engine named Kadabra. This weaver is able to manipulate Java code allowing to easily restructure the source code to prepare it for the runtime adaptivity features. To effectively add runtime adaptivity features in an application, a Java API with runtime adaptivity features was developed. This Java API includes functionalities that support runtime autotuning and adaptivity actions, in the context of runtime adaptivity strategies specified in LARA, including features for template-based code generation. The template-based code generator can replace the bytecodes of a method or generate new methods. A LARA API was also developed to facilitate the integration of LARA strategies with the Java API for runtime adaptivity.

The use of the LARA, API, and source-to-source compilation avoids mistakes on code transformations and allows the programming of reusable strategies in terms of their use by other strategies for the same target and for other target applications. With LARA strategies, it is possible to try different approaches and different search schemes and keep the source code clean.

The approach was presented in the context of different case studies. A set of runtime adaptivity strategies for those case studies were programmed in the LARA language, showing the common approach for the specification of runtime adaptivity strategies. Then, an evaluation of
Conclusions

the proposed runtime adaptivity strategies is presented, where the results highlight the importance of the approach when applying runtime adaptation. The experimental results show that runtime decisions improve performance, especially on long-time running applications, and code specialization also contributes to performance improvements. The evaluation shows that the approach is able to improve the performance of generic method versions by replacing them with specialized versions at runtime. The autotuning features facilitate the exploration of different possibilities and allow the specification of the exploration scheme. With respect to the runtime autotuning schemes used for evaluating the approach, the overhead of the runtime autotuning is very small and almost insignificant.

The evaluation of the programming effort of runtime adaptivity strategies shows a potential increase in productivity, compared to a less automated approach or an approach considering the use of multiple tools. For the latter, the LARA framework provides mechanisms of integrating external tools allowing a unified approach if one intends to use different tools. The use of actions and features available as LARA APIs eases the effort of writing adaptivity strategies and improves reusability. Highly reusable aspects can be used for different versions of the same application or be used in other target applications. Furthermore, the target-language agnosticism of the LARA language potentially allows the use of those reusable aspects over applications developed in a different target language, considering the existence of a runtime adaptivity API in that target language.

8.2 Future Work

The ongoing and future work regarding the programming and mapping strategies for embedded runtime adaptivity includes the improvement of the developed framework and the study of new research paths. Future work should consider more advanced refactoring techniques to prepare the application code for runtime adaptivity schemes, and the extension of the weaving framework with more functionality that may improve the development of adaptivity strategies.

The LARA framework is still missing traceability features. It is still necessary schemes to trace the join point visiting patterns on the target application, the changes applied during code transformations and the impact of those changes in both target application and the executing LARA aspects. A transformation might change the structure of the application that invalidates other program locations, producing, for instance, dead code. This also influences the validity of the selected join points as they might become invalid references to the program tree. Another problem is the traceability of aspects over different versions of the application. Since programs can be updated regularly, it is important to have ways to validate the aspects that have been continuously applied in the application in order to verify if they are still behaving as expected, i.e., they are still achieving the same goal as initially stated.

The runtime adaptivity framework can still be further improved, including the development of other autotuning schemes. A high-level specification of templates for code generation is a deserved research path that was not addressed in this thesis. An approach would consider the design of using a DSL based on the high-level target programming language annotated with the code transformations to be applied at runtime and the expected input arguments for the generation. A compiler for this DSL would translate the code into a template that could be used in the current template-based runtime generation approach. The adaptivity API requires mechanisms for the
Conclusions

verification of the runtime adaptivity features. Currently, the verification and validation of the runtime adaptivity behavior have to be programmed and a more automated and reliable approach should be researched. This should include the compile-time validation of the execution points and the adaptivity strategy and the runtime verification to check if the program is executing normally and none undesired side effects occur due to the runtime adaptivity strategy. The runtime verification should also check if the adaptivity is behaving as expected. The runtime adaptivity API should also consider the support for more advanced autotuning approaches such as the one provided by the mARGOt autotuner (Gadioli et al., 2019).

Another research path is the extension of the proposed approach to provide runtime adaptivity to different target languages and/or domains. An example of this work being integrated into the C++ language can be seen in (Bispo et al., 2019). Another example is to fully extend the approach to Android applications, studying the changes of ahead-of-time compilers and the possible runtime scenarios and changes the adaptivity strategies should deal with.

Future work should also consider the development of a DSL with higher levels of abstraction that would further improve the developer experience in the specification of runtime adaptivity, although with less power and flexibility on the strategy design. The proposed infrastructure could be used as a bridge between an easy, more automated, development of runtime adaptivity strategies with less maneuverability for the developer and a semi-automated approach and with additional flexibility. Another aspect to consider is the research and development of DSLs using the LARA language and Kadabra infrastructure as a middleware to other domains.

The presented case studies depicted the use of the proposed approach in different case scenarios in which runtime adaptivity was important. However, the approach should be evaluated with additional, more complex applications and/or complex adaptivity strategies to further demonstrate the potential of an approach providing the programming of runtime adaptivity strategies while maintaining the complex source code intact. An example is the use of the proposed approach in the context of internet of things (IoT) and over the various IoT layers, from in-situ computations to cloud computing should also be considered. For example, this approach has the potential of providing strategies for runtime adaptivity at the edge level and that, on certain runtime scenarios, offload computations from the edge level to cloud-based services.

The continuous promotion of the proposed approach is an ongoing work. This includes, besides writing scientific publications, writing documentation regarding the approach and the related developed tools and infrastructure and the preparation and presentation of tutorials and demos, including online materials. Two examples of these online materials are the LARA wiki\textsuperscript{13} and the Kadabra online demo\textsuperscript{14}. The development of a forum in which developers could add questions, proposals and tips regarding both LARA language and the specification of runtime adaptivity could develop a community that helps in the promotion and the improvement of the proposed approach.

\textsuperscript{13} LARA wiki: https://fe.up.pt/~specs/projects/lara/doku.php
\textsuperscript{14} Kadabra online demo: https://specs.fe.up.pt/tools/kadabra
Conclusions
References


References


Gamma, E. (1995). Design Patterns: Elements of Reusable Object-Oriented Software Addison-Wesley.


References


References


References


References


This appendix lists the benchmarks used in the experiments conducted in this work. Different types of algorithms were tested, namely in the area of image and signal processing and matrix operations. The benchmarks are written in Java™ language.

A.1 Image Smooth Algorithm

Consider the image smoothing algorithm in Figure A.1. This method receives as input a gray image and outputs a smoothed version of that image. Each pixel is smoothed according to the surrounding neighbors. Each neighbor has a specific weight, and the smoothed value is measured as the median of the weighted pixel values. A squared window with the target pixel in the middle is used to select those neighbors. The size of this window is variable, with the weight of the neighbors associated with each cell in a window. The window, predefined as 3 by 3 in this example, is defined prior to calling the smooth method. Moreover, the window size does not change during the method execution. It is used in lines 8-10, where two loops iterate according to the window size (lines 8 and 9) and an assignment requires the access to a position of the window (line 10). The iterated values in these loops also affect accesses to the input array (line 10).

```java
short[][] K = { { 1, 2, 1 }, { 2, 4, 2 }, { 1, 2, 1 } };  
short WX = 3;  
short WY = 3;  
void smooth(short[][] IN, short[][] OUT, int i, int j) {  
    for (int row = 0; row < i - WX + 1; row++) {  
        for (int col = 0; col < j - WY + 1; col++) {  
            int sumval = 0;  
            for (int wrow = 0; wrow < WX; wrow++) {  
                for (int wcol = 0; wcol < WY; wcol++) {  
                    sumval += IN[row + wrow][col + wcol] * K[wrow][wcol];  
                }  
                sumval = sumval / 16;  
                OUT[row][col] = (short) sumval;  
            }  
        }  
    }  
```

Figure A.1. A weighted sliding-average smooth algorithm, using a WX by WY window.
Benchmarks

In this method, at least one program specialization can be applied by using the window information. For example, using the window size, and corresponding constants, some optimizations, such as loop unrolling and constant propagation, can be used and chained to obtain an improved code version. Having knowledge of the window size, as exemplified in Figure A.2, the loops that iterate over the window matrix can be fully unrolled, and the array accesses can be replaced with the corresponding constants (constant propagation). This provides an optimized version of the smooth algorithm, without the overhead of the two innermost loops and the array accesses regarding the loading of the coefficients store in array K.

```c
void smooth(short[][] IN, short[][] OUT, int i, int j) {
    for (int row = 0; row < i - 4; row++) {
        for (int col = 0; col < j - 4; col++) {
            int sumval = 0;
            sumval += IN[row][col] * 1;
            sumval += IN[row][col + 1] * 2;
            sumval += IN[row][col + 2] * 1;
            sumval += IN[row + 1][col] * 2;
            sumval += IN[row + 1][col + 1] * 4;
            sumval += IN[row + 1][col + 2] * 2;
            sumval += IN[row + 2][col] * 1;
            sumval += IN[row + 2][col + 1] * 2;
            sumval += IN[row + 2][col + 2] * 1;
            sumval = sumval / 16;
            OUT[row][col] = (short) sumval;
        }
    }
}
```

Figure A.2. Example of a specialized version of the smooth algorithm, for a 3×3 window.

Figure A.3 shows some speedups attainable when specializing the smooth algorithm with three different strategies, for two different systems (x64 PC and an Odroid), with different figure sizes\textsuperscript{15}. In the first strategy (LU) the two innermost loops of the algorithm are fully unrolled for a given window size of X. The second strategy (LU+CP) uses the first strategy (LU) and replaces the accesses to the array K with the corresponding constant value. Figure A.2 is an example of generating a new code version with this strategy, where X=3. The third strategy uses the second strategy (LU+CP) and adds functionality that temporarily stores the last X-1 columns. These strategies are able to achieve speedups between 4.20 to 5.50 for the x64 PC and 3.50 to 4.30 for Odroid.

\textsuperscript{15} For more motivational examples please see Appendix B
Figure A.3. Speedups achieved for the median smooth algorithm with three specialization approaches: full loop unroll (LU), LU with constant propagation (LU+CP), LU+CP with data reuse (LU+CP+DR).

A.2 FIR

The FIR (Finite Impulse Response) method is a filter used in digital signal processing, using a coefficients window of size \( N \) to process a given sample, having as reference the previous \( N-1 \) samples. As input, the method receives an array of samples for processing and an array, of size \( N \), with the coefficients given for each position of the window. The method then processes each position from the input samples with the given window of coefficients, by calculating a weighted mean between the processing position and the previous \( N-1 \) elements. FIR outputs an array containing the weighted-means.

```java
public static int[] fir(int[] x, int[] c) {
    int[] y = new int[x.length];
    int M = x.length;
    int N = c.length;
    for (int j = N - 1; j < M; j++) {
        int output = 0;
        for (int i = 0; i < N; i++) {
            output += c[i] * x[j - i];
        }
        y[j] = output;
    }
    return y;
}
```

A.3 Matrix Multiplication

The matrix multiplication algorithm produces the product between: matrix*matrix, matrix*vector or vector*matrix. In the preliminary experiments conducted only the matrix*matrix multiplication was considered. The following algorithm outputs a matrix containing the product between the two input matrices.
A.4 Weighted-Average Smooth

A weighted sliding-average smooth algorithm for image processing outputs a smoothed version of a given input image, where each pixel is smoothed according to the surrounding neighbors. A squared window with the target pixel in the middle is used to select those neighbors. Each neighbor has a specific weight, and the smoothed value is measured as the mean of these pixels.

```
short[][] K = { { 1, 2, 1 }, { 2, 4, 2 }, { 1, 2, 1 } };  
short WX = 3;  
short WY = 3;  
void smooth(short[][] IN, short[][] OUT, int i, int j) {  
    for (int row = 0; row < i - WX + 1; row++) {  
        for (int col = 0; col < j - WY + 1; col++) {  
            int sumval = 0;  
            for (int wrow = 0; wrow < WX; wrow++) {  
                for (int wcol = 0; wcol < WY; wcol++) {  
                    sumval += IN[row + wrow][col + wcol] * K[wrow][wcol];  
                }  
            }  
            sumval = sumval / 16;  
            OUT[row][col] = (short) sumval;  
        }  
    }  
}
```

A.5 Median Smooth

The median smooth algorithm calculates the output value of each processed pixel based on the median of the N neighbors (where N = M*M, for an MxM window) of the corresponding input pixel. The program uses a sliding window of size MxM and, for each pixel, gets the neighbors in the window and calculates the median value by sorting the elements and retrieving the middle value. The following code depicts the median smooth algorithm from the Hipr2 benchmark.

```
short[][] K = { { 1, 2, 1 }, { 2, 4, 2 }, { 1, 2, 1 } };  
short WX = 3;  
short WY = 3;  
void smooth(short[][] IN, short[][] OUT, int i, int j) {  
    for (int row = 0; row < i - WX + 1; row++) {  
        for (int col = 0; col < j - WY + 1; col++) {  
            int sumval = 0;  
            for (int wrow = 0; wrow < WX; wrow++) {  
                for (int wcol = 0; wcol < WY; wcol++) {  
                    sumval += IN[row + wrow][col + wcol] * K[wrow][wcol];  
                }  
            }  
            sumval = sumval / 16;  
            OUT[row][col] = (short) sumval;  
        }  
    }  
}
```
Benchmarks

```java
public int [][] smooth(int [][] input, int [][] kernel,
                      int width, int height,
                      int iterations){
    int [][] outputArrays = new int [width][height];
    for (int its=0;its<iterations;++its){
        for(int j=0;j<height;++j){
            for(int i=0;i<width;++i){
                outputArrays[i][j] = medianNeighbour(input,kernel,width,height,i,j);
            }
        }
        input = (int [][]) outputArrays.clone(); //copy output to input
    }
    return outputArrays;
}
```

```java
public int medianNeighbour(int [][] input, int [][] kernel,
                           int w, int h, int x, int y) {
    ArrayList values = new ArrayList();
    for(int j=0;j<3;++j){
        for(int i=0;i<3;++i){
            if((kernel[i][j]==1) && ((x-1+i)>=0)
               && ((y-1+j)>=0) && ((x-1+i)<w) && ((y-1+j)<h )){
                values.add(new Integer(input[x-1+i][y-1+j]));
            }
        }
    }
    int m = getMedian(values);
    return m;
}
```

```java
public int getMedian(ArrayList values){
    int median;
    if(values.size()/2 == (values.size()+1)/2){ //if even number of elements
        for(int i=0;i<(values.size()-1)/2;++i){
            removeMax(values);
        }
        median = getMax(values); //median is the mean of two central values
    } else{ //if odd number of elements
        for(int i=0;i<values.size()/2;++i){
            removeMax(values);
        }
        median = getMax(values); //median is central value
    }
    return median;
}
```

A.6 Median by Sort Algorithms

The following methods depict the sorting algorithms used to replace the getMedian method, replacing the output of the completed sorted array with the return of the median value.
Benchmarks

Odd-Even Sorting Network

```java
public int sortNet(int[] values, int length) {
    int init = 0; // for odd-even swaps
    for (int i = 0; i < length; i++) {
        for (int index = init; index < length - 1; index += 2) {
            if (values[index] > values[index + 1]) {
                int temp = values[index];
                values[index] = values[index + 1];
                values[index + 1] = temp;
            }
        }
        init = 1 - init; // for odd-even swaps
    }
    if (length % 2 == 0) { // length is even
        int middleLeft = values[length / 2 - 1];
        int middleRight = values[length / 2];
        return (middleLeft + middleRight) / 2;
    }
    return values[length / 2];
}
```

Counting Sort

```java
public int getMedian(int[] values) {
    int[] sortingTable = new int[256];
    for (int i = 0; i < values.length; i++) {
        sortingTable[values[i]]++;
    }
    int sum = 0;
    int median = 0;
    if (values.length % 2 == 0) {
        int first = 0;
        int second = 0;
        int divS = values.length / 2;
        int divB = divS + 1;
        while (sum < divS) {
            sum += sortingTable[first];
            first++;
        }
        second = first;
        while (sum < divB) {
            sum += sortingTable[second];
            second++;
        }
        first--;
        second--;
        median = (first + second) / 2;
    } else {
        int div = values.length / 2 + 1;
        while (sum < div) {
            sum += sortingTable[median];
            median++;
        }
        median--;
    }
    return median;
}
```
A.7 Sobel

Sobel is an edge detection algorithm that consists of three phases: 1) a Gaussian convolution to smooth the input (gray) image, 2) convolution with a vertical Sobel kernel and 3) convolution with a horizontal Sobel kernel.
Benchmarks

```java
public int[][] sobel(int[][] inputImage) {
    int M = inputImage.length; int N = inputImage[0].length;
    int[][] vertical; int[][] horizontal; int[][] smoothed;
    int[][][] filter = new int[K][K];
    /* Perform the Gaussian convolution. */
    filter = { {1,2,1}, {2,4,2}, {1,2,1} };
    smoothed = convolve2d(inputImage, filter);
    /* Convolve the smoothed matrix with the vertical Sobel kernel. */
    filter = { {1,0,-1}, {2,0,-2}, {1,0,-1} };
    vertical = convolve2d(smoothed, filter);
    /* Convolve the smoothed matrix with the horizontal Sobel kernel. */
    filter = { {1,2,1}, {0,0,0}, {-1,-2,-1} };
    horizontal = convolve2d(smoothed, filter);
    /* Take the larger of the magnitudes of the horizontal and vertical */
    Form a binary image by comparing to a threshold T.
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; ++j) {
            int temp1 = Math.abs(vertical[i][j]);
            int temp2 = Math.abs(horizontal[i][j]);
            int temp3 = (temp1 > temp2) ? temp1 : temp2;
            smoothed[i][j] = (temp3 > T) ? 255 : 0;
        }
    }
    return smoothed;
}
```

The convolution method, convolve2d, takes as input an image and a kernel containing the coefficients to use over the target image. It starts by calculating the normal factor with the input kernel (lines 6 to 10), and then, for each pixel, calculates the convolution by means of the input kernel.

```java
public static int[][] convolve2d(int input_image[][], int kernel[][]){
    int N = input_image.length; int M = input_image[0].length;
    int out_image[][] = new int[N][M];
    int dead_rows = K / 2; int dead_cols = K / 2;
    int normal_factor = 0;
    for (int r = 0; r < K; r++) {
        for (int c = 0; c < K; c++) {
            normal_factor += abs(kernel[r][c]);
        }
    }
    for (int r = 0; r < N - K + 1; r++) {
        for (int c = 0; c < M - K + 1; c++) {
            int sum = 0;
            for (int i = 0; i < K; i++) {
                for (int j = 0; j < K; j++) {
                    sum += input_image[r + i][c + j] * kernel[i][j];
                }
            }
            out_image[r + dead_rows][c + dead_cols] = sum / normal_factor;
        }
    }
    return out_image;
}
```

Appendix B

Preliminary Results

This appendix contains a set of motivational experiments. It shows some speedups achieved with preliminary experiments of runtime adaptivity by means of code specialization. The code specialization is achieved applying compiler optimizations based on runtime information. To remark the feasibility and advantage of using runtime adaptation, several algorithms that target different areas were tested, namely signal, image and matrix processing. The experiments were conducted applying several types and sequences of optimizations. The results mainly intend to demonstrate that:

i. Code specialization may optimize an application;

ii. Different circumstances require different approaches;

iii. Different target system requires different specialization strategies;

iv. The overhead imposed by runtime adaptation may influence program efficiency.

B.1 Experimental Setup

The experiments present in the following sections were conducted with different program contexts. This allows us to observe the practicability of the runtime adaptivity approach within different contexts. The proposed approach focuses on Java versions of the applications, in order to apply the code specialization approach regarding the JVM. The target benchmarks are described in Table B-I. Six algorithms were selected to test different specialization strategies: an algorithm for signal processing (FIR), a matrix multiplication operation, and three algorithms for image processing.
Preliminary Results

Table B-I. Characteristics of the Benchmarks used in the experiments.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
<th>Characteristics</th>
<th>Runtime Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIR</td>
<td>Filter used in digital signal processing,</td>
<td>Coefficient array of size N</td>
<td>Coefficients array</td>
</tr>
<tr>
<td></td>
<td>Inner loop to iterate the coefficients</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>Multiplication operation taking two matrices to output a third one.</td>
<td>Includes three nested loops to process both matrices</td>
<td>Matrices size</td>
</tr>
<tr>
<td>Weighted-Average Smooth</td>
<td>Filter used for image smoothing, based on a weighted mean.</td>
<td>Coefficient array of size NxN</td>
<td>Coefficients array</td>
</tr>
<tr>
<td></td>
<td>Two Inner loops to iterate the coefficients</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Median Smooth</td>
<td>Filter used for image smoothing, based on the median value.</td>
<td>Different algorithms may be used to calculate the median</td>
<td>Number of neighbors</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sobel</td>
<td>Edge detection process divided into three convolution phases:</td>
<td>Coefficient array of size NxN</td>
<td>Coefficients array</td>
</tr>
<tr>
<td></td>
<td>gaussian, vertical and horizontal.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Two Inner loops to iterate the coefficients</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Different optimizations/specializations were observed to test different specialization strategies opportunities for each program, where these strategies have some dependency on some information only available at runtime. Furthermore, different specialization phases were tested in order to observe the execution behavior with the optimizations applied in each version. For instance, for the FIR method, using a window of size 4, a set of specialized versions were created taking into account the window size and the constants related to that window.

Table B-II depicts the strategies considered for each target algorithm. Each strategy considers a set of optimizations. Some of the most relevant optimizations applied are: algorithm selection (AS), loop unroll (LU), loop interchange (LI), loop tiling (LT), scalar replacement (SR), data reuse (DR), constant propagation (CP) and function inlining (FI).
### Preliminary Results

Table B-II. Selected specialization experiments for each algorithm.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Specialization Description</th>
<th>Target method</th>
<th>Optimization Chain</th>
<th>Runtime Information</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FIR</strong></td>
<td>a) Fully unroll the innermost loop</td>
<td></td>
<td>LU</td>
<td>Coefficients array size</td>
</tr>
<tr>
<td></td>
<td>b) specialization a) plus coefficients propagation</td>
<td></td>
<td>LU + CP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c) specialization b) plus data reuse according to the array size</td>
<td>fir</td>
<td>LU + CP + DR</td>
<td>Coefficients array size and values</td>
</tr>
<tr>
<td></td>
<td>d) specialization b) plus loop unroll, with a factor of 2, of the outermost loop</td>
<td></td>
<td>LU + CP + LU</td>
<td></td>
</tr>
<tr>
<td><strong>Matrix Mult.</strong></td>
<td>a) Replace writes to the output array with a scalar variable and store only once.</td>
<td>SR</td>
<td></td>
<td>n.a.</td>
</tr>
<tr>
<td></td>
<td>b) Interchange the loop iterating j with the loop iterating k</td>
<td>multiply</td>
<td>LI (ikj)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c) specializations a) plus b)</td>
<td></td>
<td>LI (ikj) + SR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d) specialization b) plus loop interchange between i and k</td>
<td></td>
<td>LI (kij)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e) Loop interchange i with j</td>
<td></td>
<td>LI (jik)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f) Loop tiling according to the matrices size</td>
<td></td>
<td>LT</td>
<td>Matrices size</td>
</tr>
<tr>
<td><strong>Weighted-Average Smooth</strong></td>
<td>a) Fully unroll the two innermost loops</td>
<td>smooth</td>
<td>LU</td>
<td>Coefficients array size</td>
</tr>
<tr>
<td></td>
<td>b) specialization a) plus coefficients propagation</td>
<td></td>
<td>LU + CP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c) specialization b) plus data reuse according to the array size</td>
<td></td>
<td>LU + CP + DR</td>
<td>Coefficients array size and values</td>
</tr>
<tr>
<td><strong>Median Smooth</strong></td>
<td>a) Select best algorithm</td>
<td>getMedian</td>
<td>AS</td>
<td>Number of neighbors</td>
</tr>
<tr>
<td></td>
<td>b) Apply optimizations depending on the algorithm selected in a)</td>
<td>getMedian +</td>
<td>AS + Spec</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c) Inline best median algorithm inside smooth</td>
<td>smooth</td>
<td>AS + Spec + F</td>
<td></td>
</tr>
<tr>
<td><strong>Sobel</strong></td>
<td>a) Fully unroll the innermost loops</td>
<td>convolve2d</td>
<td>LU</td>
<td>Coefficients array size</td>
</tr>
<tr>
<td></td>
<td>b) Create multiple convolution methods and move the filters inside</td>
<td>sobel +</td>
<td>-</td>
<td>Coefficients array size and values</td>
</tr>
<tr>
<td></td>
<td>c) a) plus b) plus coefficient propagation</td>
<td>convolve2d</td>
<td>LU + CP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d) c) + data reuse</td>
<td></td>
<td>LU + CP + DR</td>
<td></td>
</tr>
</tbody>
</table>

It is challenging to achieve an accurate execution time as the system where the experiments were executed are not devoted to that task, and system tasks are executing parallel to the tests, among other possible issues. For more accurate results, each test set was executed several times, discarding the first few executions, considered as a warm-up for the JVM to study and produce JIT compiler optimizations. The remaining execution defines the average execution time.

The experiments were executed in a PC with Ubuntu x64 14.04 LTS, Intel® Quad Core™ i5 CPU 650 @ 3.20 GHz, with 8 GB DDRAM. The Java Virtual Machine used was a Java(TM) SE Runtime Environment (build 1.7.0_55), and the version 1.7.0_55 of javac was used for compiling Java applications. Some of these tests were also executed in a different environment, namely an Odroid system, Cortex™-A15 2.0 GHz quad-core and Cortex™-A7 quad-core CPU. This allows us to observe how different optimizations applied behaves with the target system.

The specification of the strategies for each experiment was based on the handcraft development of a Java Agent (responsible to execute the strategy), a Java class containing the template for code generation, and an instrumenting class implementing the java.lang.instrument interface. The target application was instrumented with invocations...
Preliminary Results

to the Java Agent to request the runtime adaptations. The agent receives the runtime adaptation request, generates the required version with the input of the event and requests the classloader to change the target class(es) with the generated version. In these experiments, Jasmin (Meyer, 2004) was used to build the Java bytecodes for the template-based code generation. The input to Jasmin is generated by Java code that implements how the code needs to be generated according to the specific template and to the given inputs.

As the performance can be influenced according to the JIT compiler optimizations, some studies were performed with the JIT optimizations turned on and off. The overhead required for the code generation and class instrumentation/redefinition were also considered and discussed during the analysis of these tests.

B.2 Results

This section shows the speedup results achieved with runtime code specialization, for different target application. These results intend to demonstrate the benefits of the runtime specialization approach that uses runtime code generation based on the use of templates. Java code was used in order to simplify how the specialized versions are generated, instead of Bytecode instructions.

B.2.1 FIR Algorithm

The FIR (Finite Impulse Response) (Rabiner and Gold, 1975), depicted in Figure B.4, is a filter used in digital signal processing, using a coefficients window of size $N$ to process a given sample, having as reference the previous $N-1$ samples. As input, the method receives an array of samples for processing and an array, of size $N$, with the coefficients given for each position of the window. In lines 7 and 8 the window is used to process the current signal with the previous signals. The window size and values do not change inside the given method. On the other hand, it influences the expected output.

```java
public static int[] fir(int[] x, int[] c) {
    int[] y = new int[x.length];
    int M = x.length;
    int N = c.length;
    for (int j = N - 1; j < M; j++) {
        int output = 0;
        for (int i = 0; i < N; i++) {
            output += c[i] * x[j - i];
        }
        y[j] = output;
    }
    return y;
}
```

Figure B.4. The FIR method in Java code.

A specialization opportunity, was found in this method, as the inner-loop can be specialized for the given coefficients window (argument ‘c’ for the fir method), applying optimizations with factors depending on the window size and the corresponding coefficients. The most relevant optimizations for this strategy (loop unroll, constant propagation and data reuse) were selected and applied in different sequences, to observe the impact of each optimization.
Figure B.4 shows the specialization strategies with the applied optimizations, considering a coefficients window of size 4, with values {2, 4, 4, 2}. Since the length of the coefficients window is known, the strategy starts by a) fully unrolling the innermost loop. Then in b) the array accesses are replaced with the corresponding constant value. Then, since each iteration of the remaining loop requires the use of the previous three values, c) these values are reused for the next iteration. The last strategy, instead of reusing values, d) it applies another loop unroll only with a factor of two, keeping the array accesses to the same position as close as possible.

```java
int[] fir(int[] x, int[] c){
    int[] y = new int[x.length];
    int M = x.length;
    int N = c.length;
    for(int j = N - 1; j < M; j++){
        int output = 0;
        output += c[0] * x[j];
        output += c[1] * x[j - 1];
        output += c[2] * x[j - 2];
        output += c[3] * x[j - 3];
        y[j] = output;
    }
    return y;
}
```

(a) Full unroll of inner loop

```java
int[] fir(int[] x, int[] c){
    int[] y = new int[x.length];
    int M = x.length;
    int N = c.length;
    for(int j = 3; j < M; j++){
        int x_3 = x[j];
        int output = 0;
        output += 2 * x_3;
        output += 4 * x[j - 1];
        output += 4 * x[j - 2];
        output += 2 * x[j - 3];
        y[j] = output;
    }
    return y;
}
```

(b) Coefficients propagation

```java
int[] fir(int[] x, int[] c){
    int[] y = new int[x.length];
    int M = x.length;
    int N = c.length;
    for(int j = 3; j < M; j++){
        int x_0 = x[0];
        int x_1 = x[1];
        int x_2 = x[2];
        int x_3 = x[j];
        int output = 0;
        output += 2 * x_3;
        output += 4 * x_2;
        output += 4 * x_1;
        output += 2 * x_0;
        y[j] = output;
        x_0 = x_1;
        x_1 = x_2;
        x_2 = x_3;
    }
    return y;
}
```

(c) Data reuse

```java
int[] fir(int[] x, int[] c){
    int[] y = new int[x.length];
    int M = x.length;
    int j = 3;
    int output2 = 0;
    int output = 0;
    output = 2 * x[j];
    output += 4 * x[j - 1];
    output += 4 * x[j - 2];
    output += 2 * x[j - 3];
    y[j] = output;
    for(j = 4; j < M; j += 2) {
        output2 = 2 * x[j + 1];
        output = 2 * x[j];
        output2 += 4 * x[j];
        output += 4 * x[j - 1];
        output2 += 4 * x[j - 1];
        output += 4 * x[j - 2];
        output2 += 2 * x[j - 2];
        output += 2 * x[j - 3];
        y[j] = output;
        y[j + 1] = output2;
    }
    return y;
}
```

(d) Unroll with a factor of 2

Figure B.5. Specialized versions of FIR, for a window of size 4 (coefficients: {2,4,4,2}).

Following the specialization decision of optimizing this method according to the given input window, a series of different tests were considered. The set of tests are to consider the execution of the application with different window sizes, where the sizes 4, 8 and 32 were selected.

The speedups obtained are depicted in Figure B.6, for the two target systems (PC and Odroid). The used strategies are generating the required versions at runtime. For instance, the
Preliminary Results

The speedups obtained for the full unroll version includes the runtime generation of the unfolded version of FIR. The x64 PC is able to achieve a speedup of $2.64 \times$ with full unroll $\Rightarrow$ constant propagation $\Rightarrow$ data reuse, for a window of size 4. Other window sizes are still able to achieve better performance with the full unroll $\Rightarrow$ constant propagation $\Rightarrow$ loop unroll(2) strategy, although with lower values. As for the execution in the Odroid system, the strategies achieve higher speedups (from 2.49$\times$ to 3.50$\times$) compared to the x64 PC. The best strategy, for a window of 8, for the x64 PC is not the same for the Odroid, as in this system the best approach is full unroll $\Rightarrow$ constant propagation $\Rightarrow$ data reuse. An interesting observation in these results is the inverted speedup results, comparing to x64, as the window size grows. Instead of having the speedup degraded, the Odroid can achieve high speedups, even with approaches that reduce the performance in the x64 PC (e.g., data reuse strategy for a window of size 32).

![Figure B.6. Speedups achieved with the specialized version of FIR, in two different target systems, for different window sizes.](image)

These results show the parameter and the target platform dependency on the strategies. The coefficients window parameter allows us to select the best optimization sequence and use its size and the coefficients to apply the optimizations more efficiently. Furthermore, knowing the platform in which the application is executing helps to select the best specialization approach, as, in this example, the best version for the Odroid may not be suitable for the x64 PC.

**B.2.2 Matrix Multiplication**

The following experiment uses the matrix multiplication operation presented in Figure B.7. In terms of contextual information, the input arguments may provide significant information about which optimizations, and corresponding optimization factor(s), should be applied.
Preliminary Results

```java
public static int[][] mult(int[] a, int[] b) {
    int rowsInA = a.length;
    int columnsInA = a[0].length; // == rows in B
    int rowsInB = b[0].length;
    int[][] c = new int[rowsInA][columnsInB];
    for (int i = 0; i < rowsInA; i++) {
        for (int j = 0; j < columnsInB; j++) {
            for (int k = 0; k < columnsInA; k++) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    return c;
}
```

Figure B.7. A Java implementation of a matrix multiplication operation.

The strategy for specialization uses loop interchange, loop tiling and scalar replacement optimizations for the experiments. These experiments were intended to observe how loop interchange and loop tiling optimizations perform with different executing systems. Figure B.8 and Figure B.9 present the speedups achieved respectively for x64 PC and Odroid systems. Different tile sizes for loop tiling were tested, from 8 to 1024. Also, three types of loop interchange were tested to see which version would be the most promising for each system: interchange iterator \( j \) with iterator \( k \) (LI_IKJ); interchange \( j \) with \( k \), followed by the interchange of \( i \) with \( k \) (LI_KIJ); and interchange \( j \) with \( I \) (LI_JIK). The loop interchange versions were pre-compiled before execution, while the use of loop tiling includes the runtime definition of the block size.

For the x64 PC (Figure B.8) the experiments are able to achieve significant speedups, scaling almost 40\( \times \) without loop tiling, and up to 80\( \times \) when considering loop tiling with a block of 512 whenever the matrices sizes allow it. The combination of scalar replacement with loop interchange between iterator \( j \) and iterator \( k \) is the best approach when a loop tiling equal or greater than 32 is not possible. For the other cases loop tiling provides slighter improvement. Hence, for the x64 PC there is a dependency on the input parameters.

![Figure B.8. Speedups achieved for the matrix multiplication in an x64 PC, with (right) and without (left) loop tiling. Matrices sizes of N\times N.](image)

The Odroid system (Figure B.9) does not achieve such significant speedups, as the top factor obtained is in almost 10\( \times \). Loop tiling does not provide execution improvements, hence being more advantageous to ignore the optimization for this system. There is no data dependency here,
as loop interchange and scalar replacement are not dependent on the input size and are not
parameterizable optimizations.

So, the strategy to be assigned depends on the target system. After knowing the target system
there may or may not be data dependency. As for the different loop interchanges tested, there was
a difference between the two platforms, with the best interchange shared by both.

Figure B.9. Speedups achieved for the matrix multiplication in Odroid, with (right) and without (left) loop
...
The convolution method is presented in Figure B.11. This method takes as input an image and a kernel containing the coefficients to use over the target image. It starts by calculating the normal factor with the input kernel (lines 6 to 10), and then, for each pixel, calculates the convolution by means of the input kernel. One of the greatest dependencies on this code is the input coefficients kernel, namely in lines 6 to 10 and 15 to 19. So, this convolution method may be specialized according to the input kernel used and different versions for each kernel must be produced.

```
public static int[][] convolve2d(int input_image[][], int kernel[][]){
    int N = input_image.length; int M = input_image[0].length;
    int out_image[][] = new int[N][M];
    int dead_rows = K / 2; int dead_cols = K / 2;
    int normal_factor = 0;
    for (int r = 0; r < K; r++) {
        for (int c = 0; c < K; c++) {
            normal_factor += abs(kernel[r][c]);
        }
    }
    /* Convolve the input image with the kernel. */
    for (int r = 0; r < N - K + 1; r++) {
        for (int c = 0; c < M - K + 1; c++) {
            int sum = 0;
            for (int i = 0; i < K; i++) {
                for (int j = 0; j < K; j++) {
                    sum += input_image[r + i][c + j] * kernel[i][j];
                }
            }
            out_image[r + dead_rows][c + dead_cols] = sum / normal_factor;
        }
    }
    return out_image;
}
```

Figure B.11. Excerpt of the Java version of the convolution2d method.

Different transformations were tested to verify how the program behaves. Initially, knowing the kernel size, a full loop unroll is performed on the innermost loops that iterate over the window coefficients. In the second transformation, the kernel inside the convolution method is moved to a local variable, instead of being a parameter. For this, a different method for each convolution operation was developed. The third transformation builds on top of the first transformation, with constant propagation being used to substitute the accesses to the array of coefficients with the corresponding values. Furthermore, as some coefficients equal zero, some statements such as `sum += image[x][y] * coefficient` were eliminated. Finally, the fourth transformation starts from the previous and reuses data from a given iteration to the next.

Figure B.12 shows an excerpt of the adapted version of the sobel method to invoke the specialized versions and a specialized version of convolve2d for the horizontal convolution. The sobel method does not contain the kernels anymore, since these are already hosted in each generated version. Furthermore, each call to convolve2d was replaced with a call to the corresponding, specialized, version. Besides the same observations seen in B.2 (full unroll, constant propagation and data reuse), the Horizontal convolution method (convolve2d_horizontal) contains two remarking parts: the normal_factor value was solved during generation and so immediately replaced with that value, and the unnecessary operations (which always result in zero) were removed.
Preliminary Results

Figure B.12. Code excerpt from a specialized version of sobel and the horizontal convolution.

The results for each of these transformations are presented in Figure B.13, where the last series denotes the execution of the entire Sobel algorithm, with the three operations specialized. These results show the speedups attained using dynamically generated versions. The convolution specialization for each type of operation provides good individual speedups. Simply fully unrolling the inner loops achieves more than 60% speedup. The replacement of the array accesses to the kernel with the corresponding coefficients is an important optimization in this specialization, removing the accesses overhead. This optimization provides speedups from a factor of 4 to 6.5. As mentioned before, when the coefficient is zero, the operation is ignored by the generator, which results in less arithmetic and assignment operations executed. This advantage can be observed when comparing the ifu_cp speedups between the vertical and horizontal convolution with the Gaussian convolution (gray bars in Figure B.13).
Figure B.13. Speedups achieved with code specialization for the Sobel algorithm for an image size of 1024×768 pixels.

Data reuse provides a slight boost from the previous optimization. Figure B.14 shows the reuse scheme applied in each iteration. For a 3×3 kernel, this transformation allows reusing 6 values from the previous iteration, meaning that only 3 array accesses are necessary (9 array accesses in the original version). The horizontal convolution takes advantage of not requiring the middle row of elements, hence only accessing and reusing the top and bottom rows (see Horizontal Conv. in Figure B.14). This means that four values are reused and only two array accesses are performed. In the vertical convolution, for a given iteration, the middle column is not necessary (see Vertical Conv. in Figure B.14). However, since the algorithm iterates from left to right, iteration \( i + 1 \) requires the strategy to save the middle column (as it will be used in iteration \( i + 2 \)). Therefore, the vertical convolution is not able to attain the same benefits as the horizontal convolution, requiring 3 array accesses and reusing 3 values.

Figure B.14. Data reuse in the Sobel algorithm and its implications for the vertical and horizontal convolution operations after constant propagation and code elimination.
Preliminary Results

The Sobel operation was executed with the specialized versions of the other three sub-operations (i.e., Gaussian, vertical and horizontal). By applying constant propagation (with or without data reuse) an overall speedup of $4\times$ was achieved.

The Sobel execution without the JVM Hotspot optimizations was also tested. Figure B.15 shows the speedups when executing without JIT. The best performing specialization strategy for all the operations is always $ifu_{cp\_dr}$, with speedups from $2.52 \times$ to $3.68 \times$. Even though it achieves acceptable speedups, the optimizations applied have more impact when HotSpot runtime optimizations are active. Therefore, the selected specializations potentiate even more JIT optimizations.

Figure B.15. Speedups achieved for Sobel without JIT optimizations.

B.3 Analysis

The conducted experiments allow to observe that runtime specialization, based on runtime information, can be accomplished to achieve acceptable improvements for an application. When dealing with JIT compilation, one must take care with which, and how, the optimizations are applied. By presenting the optimizations phased by individual specializations, it is possible to observe how the JIT performs with each practiced optimization.

The runtime information allows to apply optimizations with more efficiency, such as the complete unroll of loops. This information may be retrieved from program information (e.g., the Sobel experiments) or even the target platform (e.g., the matrix multiplication). Furthermore, this allows us to dynamically select the best strategy.

Dynamically selecting the best algorithm approach for an execution state allows for even further improvement instead of just adapting the original version, as seen in the experiments with median smooth. Runtime selection is an advantageous approach when considering a library with multiple algorithms for the same purposes, with ups and downs for different situations. It is even possible to further improve the performance of the selected algorithm by applying compiler optimizations, possibly with the same runtime information used for the selection.

Sobel is a decent example of a program using the same method multiple times with different input states and so a single specialization is practicable/feasible. Good improvements in the
execution can be achieved by producing multiple, specialized, versions of a method. These multiple versions may even use a different specialization strategy according to the given information, as perceived in the horizontal vs. vertical convolutions results, in which the best approach for one is not the same for the other.

As for the overhead of dynamically generating specialized versions and replace the original execution with the new version, Figure B.8.16 presents the overheads for the runtime sequential adaptation when conducting some of the previous experiments, namely median smooth and Sobel, as well as the speedup achieved, for different numbers of input values. The chosen specialized versions for these measurements are the ones that provided the best performance in the previous tests, therefore, different charts present different adaptations. The overhead is presented as a percentage of the total execution time and can be seen on the right side of the y-axis (diamond mark). The overhead is composed by the time needed to generate the specialized code and to redefine the affected classes. The left side y-axis (circle mark) shows the resulting speedup of the specialization. The speedup achieved is considering the overhead of the selection of the best algorithm, the generation of the specialized version and the runtime Bytecodes replacement.

The overhead accounts for a larger percentage of the total execution time when dealing with shorter computation times since code generation is performed only once. This can be seen when increasing the number of images or by comparing different window sizes for the same number of images. The longer the program runs, the smaller the impact of this one-time operation, e.g., in the 3×3 Smooth, the overhead drops from 49% to 0.2%. Hence, the generation of specialized code may be unsuitable when the application executes for a short duration, as is the case for the Sobel operation. The generation of specialized code takes longer than the execution of the original algorithm for a single image (50% loss of performance). Nevertheless, it is possible to observe that, in most cases, even for a small number of images the overhead is a small throwback compared to the performance gain over the original version.

The speedups for the Smooth program increase as the number of images grows. For instance, in the 7×7 Smooth example, the speedup increases from 5.61 for one image, to 10.15 for 100 images. As the number of images grows, the adaptation overhead becomes insignificant.
Preliminary Results

Figure B.8.16. Speedups achieved (left axis) and the overhead applied (right axis), for different number of input images, when specializing the Median smooth (for the three tested windows) and Sobel programs.
About the Author

Name: Tiago Diogo Ribeiro de Carvalho

Email: tdrc@fe.up.pt

Scholar: http://goo.gl/t9MPNi

LinkedIn: pt.linkedin.com/in/tiagodrcarvalho

Short Bio

Tiago D. R. Carvalho was a PhD student in the Faculty of Engineering of the University of Porto (FEUP), at the Department of Informatics Engineering (DEI). He was also a member of INESC-TEC. He obtained his MSc degree in 2011 at FEUP.

Tiago’s previous work includes the development of an Aspect-Oriented Programming language (LARA) in the context of two projects: AMADEUS, partially funded by Fundação para a Ciência e Tecnologia (FCT); and REFLECT, funded by the European Community under the Framework Programme 7 (FP7). This work provided basis and motivation for the main ideas focused on this thesis. Tiago works with compiler-related topics such as domain-specific languages and compiler optimizations. Current work focuses on runtime adaptivity and specialization following an aspect-oriented programming methodology.
PhD-Related Publications

The following publications refer to work published that is not directly related, but with some emphasis, to the PhD work.

Book Chapter


Journal Publications


Conference/Workshop Publications


Oral Presentations without Publications


Posters

About the Author


Other PhD-Relevant Publications

The following publications refer to work published that is not directly related, but with some emphasis, to the PhD program.

Journal Publications


Conference/Workshop Publications

- Ricardo Nobre, Pedro Pinto, Tiago Carvalho, João M. P. Cardoso, Pedro C. Diniz, LARA-based Strategies for Targeting Multicore Architectures, in 17th Workshop on Compilers for Parallel Computing (CPC’ 2013), Lyon, France, July 3-5, 2013.

Posters