M 2015

U. PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

INTERACTIVE TOOL FOR SPECIFYING DEDICATED HARDWARE ACCELERATORS

JOÃO PAULO PAIVA REBOCHO

DISSERTAÇÃO DE MESTRADO APRESENTADA
À FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO EM
ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

MIEEC - MESTRADO INTEGRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES 2014/2015

A Dissertação intitulada

"Interactive Tool for Specifying Dedicated Hardware Accelerators"

foi aprovada em provas realizadas em 16-07-2015

o júri [illegible]

Presidente Professor Doutor António José Duarte Araújo
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

[illegible]

Professor Doutor Gustavo Ribeiro da Costa Alves
Professor Adjunto do Departamento de Engenharia Eletrotécnica do Instituto Superior de Engenharia do Porto

[illegible]

Professor Doutor João Paulo de Castro Canas Ferreira
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projeto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extratos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são corretamente citados.

João Paulo Paiva Rebocho

Autor - João Paulo Paiva Rebocho

Faculdade de Engenharia da Universidade do Porto

Faculdade de Engenharia da Universidade do Porto

Interactive tool for specifying dedicated hardware accelerators

João Paulo Paiva Rebocho

Relatório de Projeto realizado no âmbito do
Mestrado Integrado em Engenharia Electrotécnica e de Computadores
Major Telecomunicações, Eletrónica e Computadores

Orientador: Professor João Canas Ferreira
Co-orientador: Professor João Cardoso

31 de Julho 2015

Resumo

O presente relatório tem como objetivo documentar todo o trabalho realizado durante a unidade curricular: Dissertação de Mestrado Integrado em Engenharia Electrotécnica e de Computadores. Neste documento é possível encontrar os resultados de uma primeira abordagem ao projeto, começando com uma introdução ao tema, seguido de uma revisão da literatura com o estado da arte, e terminando com uma explicação sobre a arquitetura e o algoritmo usado para desenvolver a ferramenta, bem como os resultados obtidos.

Uma investigação em curso, por Nuno Paulino consiste no mapeamento de traços binários para *hardware* em tempo de execução. O sistema desenvolvido gera descrição de *hardware* correspondente a um programa a ser executado num sistema embarcado. A sua finalidade é fazer seleções de alguns troços do programa para ser mapeado para o RPU em vez da GPU de modo a melhorar o desempenho global do sistema. Essas seleções são baseadas em grupos de instruções denominadas por *Megablocks*, que representam uma porção bastante elevada da execução total do programa.

A análise completa das computações e a deteção de Megablocks são feitos pela ferramenta *Megablock Extractor* desenvolvida por João Bispo que implementou várias técnicas complexas que permitem a partição dinâmica ao nível binário. *Megablocks* extraídos com esta ferramenta representam pedaços de código que tendem a repetir-se uma elevada quantidade de vezes e, portanto, ao usar essas unidades de deteção como elementos de estudo e aperfeiçoamento, há melhores chances de obter uma melhoria do desempenho global.

A ferramenta interativa desenvolvida nesta dissertação pode ser vista como um elo entre esses dois projetos descritos anteriormente. Ela lê as informações extraídas pelo *Megablock Extractor* e representa-as numa interface gráfica interativa de modo a permitir uma melhor visualização e compreensão dos *Megablocks*. Ela também implementa uma técnica que permite a fusão de dois Megablocks. Assim, abre a possibilidade de ter dois *Megablocks* mapeados no RPU em vez de apenas um e, portanto, reduzindo a sobrecarga de comunicação entre o GPU e o RPU.

Abstract

The present report has the goal to document all the work done under the course unit: Dissertation of Integrated Master in Engineering Electronic and Computers. It is possible to find on it the results of a first approach to the project, starting with a theme introduction, followed by a literature review prototype with the state of the art, and ending with an explanation of the architecture and the algorithm used to develop the tool as well as its results.

An investigation being held by Nuno Paulino consists on mapping binary traces to hardware in run-time. The system that he developed generates hardware description corresponding to a program to be executed on an embedded system. Its purpose is to make selections of some portions of the program to be mapped into the RPU instead of the GPU to improve the system's performance. These selections are based on groups of instructions denoted *Megablocks*, which represent a fairly high portion of the total execution of the program.

The complete analysis of computations and detection of *Megablocks* is done by the *Megablock Extractor* tool developed by João Bispo which implements several complex techniques that allow dynamic partition at the binary level. Megablocks extracted from this tool represent chunks of code that tend to repeat themselves a high amount of times and therefore, by using these detection units as elements of study and improvement, there are better chances to get better overall performance boost.

The interactive tool developed on this dissertation can be seen as a link between those two works. It reads the information extracted from the *Megablock Extractor* and represents it on an interactive GUI so that a better visualization and understanding of the *Megablocks* is obtained. It also implements a technic that allows the merge of two *Megablocks*. By doing so, it opens the possibility to have two Megablocks mapped into the RPU instead of just one and therefore reducing the communication overhead generated by moving computations between the GPU and the RPU.

Acknowledgments

I would like to thank my supervisor, João Canas Ferreira, and my second supervisor, João Manuel Paiva Cardoso, for their support and guidance during my work on the dissertation

I would also like to thank my colleagues: João Bispo and Nuno Paulino for being available to explain me their work and specially for helping me to connect my work to theirs.

Last but not least, I thank my family for all the encouragement, motivation and understanding that helped me to finish this dissertation.

Table of Contents

List of Figures

List of Tables

Acronyms and Symbols

AMBER	Adaptive Dynamic Extensible Processor
ARM	Advanced RISC Machine
CCA	Custom Compute Accelerator
CDFG	Control and Data Flow Graph
CGRA	Coarse-Grained Reconfigurable Array
DEEC	Departamento de Engenharia Electrotécnica e de Computadores
DIM	Dynamic Instruction Merging
FEUP	Faculdade de Engenharia da Universidade do Porto
FPGA	Field Programmable Gate Array
FSL	Fast Simplex Link
FU	Functional Unit
GUI	Graphical User Interface
GPU	General Purpose Unit
GPS	Global Positioning System
HDL	Hardware Description Language
MIPS	Microprocessor without Interlocked Pipeline Stages
OS	Operating System
PC	Program Counter
PDF	Portable Document File
PLB	Processor Local Bus
RFU	Reconfigurable Functional Unit
RISC	Reduced Instruction Set Computing
RPU	Reconfigurable Processing Unit
SVG	Scalable Vector Graphics
WWW	World Wide Web

Chapter 1

Introduction

1.1 - Motivation

Embedded application systems play a very important role on technological evolution as they are prevalent in our everyday life and can be found everywhere, from commercial electronic products (tablets, cell phones, televisions, microwave ovens, automobiles), to much bigger and complex systems like GPS, WWW, oil refineries and nuclear plants.

Embedded systems are becoming more and more demanding, they require better performances with lower power consumption and lower area. Researches on this matter are of great importance, extensive design automation and optimization tools are crucial to create more complex embedded systems that can perform as required.

A common practice is to enhance the performance of embedded applications executing on general purpose processors (GPP) is to map computationally intensive parts (hot-spots) to specialized hardware such as *Reconfigurable Processing Units* (RPU) that act like acceleration coprocessors of the GPP.

An interesting technique to reduce energy consumption and to improve execution time, is the dynamic mapping. This allows to move computations from the GPP to the coprocessor in a transparent and flexible way, at runtime, and without pre-changing the program binary.

Currently, there is a research being done on this matter [1] that uses dynamic mapping on an embedded system model mainly consisting of a Xilinx MicroBlaze processor connected to a Coarse Grained Reconfigurable Array (CGRA). The technique used is to optimize the code segments that are most repeated. As they represent a very big portion of the execution of a program, this will result in a great overall performance boost. The detection of these code segments is gradually made by first detecting BasicBlocks. Each structure of these is formed by a sequence of instructions with both only a single entry-point and one exit-point. By adding several BasicBlocks, a SuperBlock is formed. This type of segment also has only a single entry but multiple exits. But the most complex structure and the one that is more important to work on is the Megablock. This is built by a sequence of SuperBlocks that repeats itself at least once. Each repetition is called an *iteration* and since Megablocks tend to usually have a high number of iterations, they can be considered as hot-spots.

1.2 - Objectives and contributions

This dissertation arises from a research, currently in progress [1], as a need to have an interactive tool that allows the user to visualize the entire data flow based on Megablock representations and to manipulate them.

The mapper can make decisions that despite of being chosen as the best ones, might not result as the most beneficial performance and a human perspective could be most useful to detect particular cases and make a final tweak.

Therefore, the main objective of this dissertation is to develop an interactive application to allow the user to see a graphical representation of the data flux of the code along with some important information about it. This would allow having an overview of the entire execution flow and immediately identifying and do some specific operations on the hot-zones (Megablocks) of the code without modifying its results. As hardware designers might not have background knowledge to totally understand and decipher the binary code, this application should have the feature to convert chosen code segments (represented as Megablocks) into its hardware representation in Verilog. The application should also keep track of the changes that the user does, to allow to recover a later version of the work that was done in case of an interruption either intended by the user or from a failure of undetermined cause as without this ability, working progress would be lost which could lead to a very frustrating usage of the application.

The approach was to use the Megablock Extractor (tool that detects and operates on Megablocks) as a starting point to develop a tool capable of manipulating Megablocks further so that a better performance can be achieved. This can be done by looking at several MegaBlocks contents and analyzing them to find common sequences of operations that would allow merging Megablocks and to create another one, able to work as the ones that originated it.

Nesting View feature on the interactive GUI, was defined as an additional objective to allow the visualization of cases that have several Megablocks nested inside other Megablocks. With Nesting View, the user can understand better how all Megablocks are related and make more precise optimizations since a deeper representation of the system is available.

When a high iteration Megablock is mapped to the CGRA, the GPP will be waiting for it to end in order to continue its work. Though the CGPRA can do all the iterations way faster than the GPP, if even a few of them had been done by the GPP, it would result into a faster conclusion time for all iterations since the GPP wouldn’t be idle all the time. This same technique can be used when several CGRA units are available. It would be advantageous to have several RPUs executing iterations of the same Megablock which would results in a great reduction of the time needed to finish all iterations. In order to get this implemented in the future, another additional objective was created. It consists on making a careful analysis of the shared resources to detect Megablocks that can be processed on several hardware devices at the same time and to find a solution for the Megablocks where this could not be possible.

The work done on this dissertation provides an interactive tool which represents an overview of the Megablocks detected by the MegaBlock Extractor and also their inside view. This helps to understand how the instructions are connected and makes it easier to detect possible techniques to be used to increase the overall performance. Sections 2.6 and 2.7 show some of the techniques that were studied. Section 2.6 explains traversal techniques and it focus on Depth First Search, which was used as a base to an algorithm developed that allows the merge of two MegaBlocks: the Multipath. While section 2.7 contains some loop transformation techniques that weren't included directly at the instruction level as it is shown but represent in some way the merge ability developed in the Multipath. By using it, a Megablock is created which can execute as two Megablocks, one at a time. This can avoid significant changes on processing between the GPU and the CGRA which can represent better performances.

As chapter five demonstrates, the application was correctly implemented and shows correctly all the MegaBlocks contents, allowing also to merge them. The merged Megablock that resulted from the Multipath couldn't be verified but analysis into it show that at the very most, some debug could need to be done but the major algorithm and techniques are already implemented.

1.3 - Structure of this report

This report is organized into six chapters. The current one is the introduction and it's about how this dissertation proposal arises and its objectives. The second chapter is the state of the art, it refers the previous work done on another researches about similar subjects and has important information that was useful for the work done on this dissertation. Chapter three explains how the tool can be used, all possible actions and how they were achieved, and also the organization of the code behind the tool. The algorithm used for merging graphs is explained on the forth chapter. Chapter five is all about validation of the tool and its results. The report ends on Chapter six with a conclusion of all the work done on this dissertation.

Chapter 2

State of the Art

2.1 - Target architecture

The general target architecture considered for this work is an embedded system consisting of a GPP and a CGRA co-processor like shown on Figure 2.1.

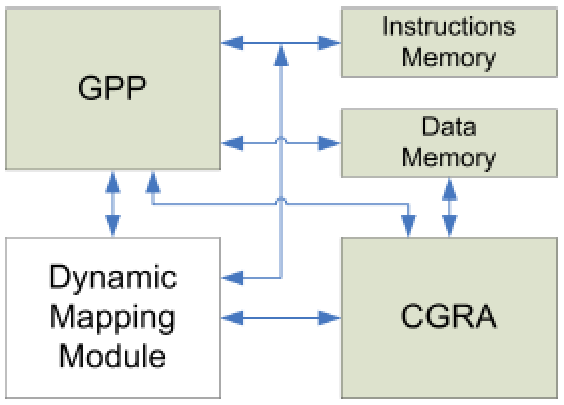


Figure 2.1 – General architecture considered [2].

The GPP that is going to be used is a MicroBlaze processor without cache and with on-chip instructions and data memories.

The CGRA model has a similar concept than the DIM architecture [3, 4], it's composed by a 2D array in which, each row can have arithmetic/logic functional units (FUs) and load/store units. With the exception of data inputs, any other communication within FUs can only be directly done to the FUs of the row below. When a FU needs to communicate with another that isn't adjacent, move instructions are used to progressively send the data along the unused FUs on each row until it reaches its destination.

The mapping module is a high-level model connected to a cycle accurate simulator for the MicroBlaze processor. The mapping algorithm is an instruction-by-instruction technique based on the one used by Clark et al. on the CCA [5, 6] in which the instructions are available for mapping in the execution order and the mapper only uses information from instructions that were previously executed. As it receives instructions and places them on the FU array, the mapper updates a table that allows it to keep track of how mapping is going and to search an empty place for the current instruction. So, each time an instruction is present to the mapper, it is placed on the first row possible according to the data dependencies and instruction restrictions. After each placement, the mapper checks if the instruction communicates with adjacent FUs and if not, it places move instructions on unoccupied FUs as needed and the routing is established. As mapping goes on, the array becomes more filled and complex which can lead to a situation where an instruction has no available row that allows a communication route with a specific FU, the mapping configuration ends and a new one starts. An example of this mapping technique is represented on Figure 1.1. It shows the sequence of instructions to be mapped, the FU array of the CGRA being filled and the register table that the mapper keeps.

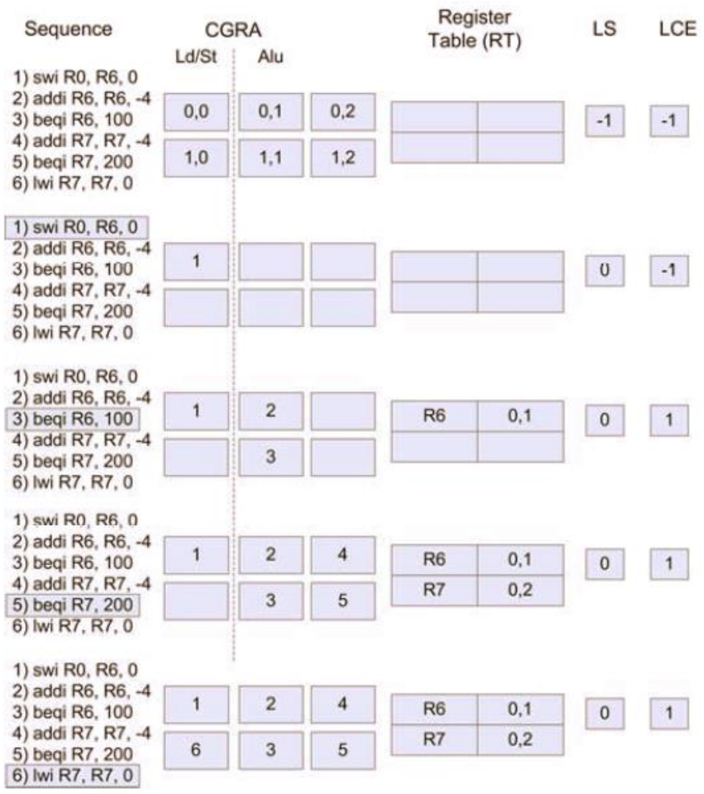


Figure 2.2 – Mapping example which shows how branches and memory operations can be mapped [2].

2.1.1 - Tests and Results

To evaluate all the work done so far, Nuno Paulino [7] has recently tested the three different architectures presented on Figure2.3 with 15 code kernels that work on 32-bit values. Tests were done with each benchmark calling its corresponding kernel 500 times.

Merge1 and merge2 are two additional tests that group together 6 kernels. For these 2 tests, several Megablocks were generated as for the other 15, they only had one Megablock implemented.

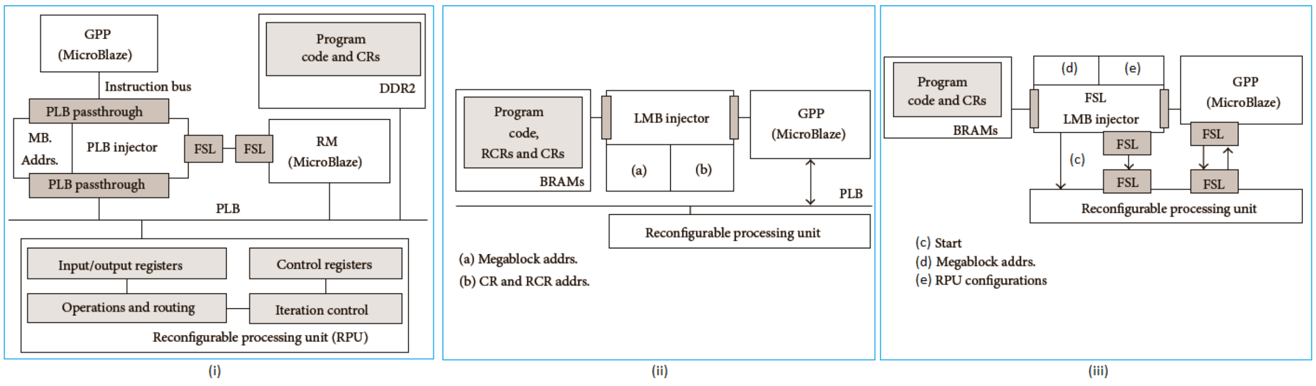


Figure 2.3 – Architectures used for testing: i) DDR-PLB; ii) LMB-PLB; iii) LMB-FSL [7].

As it can be seen on Table 2.1, the MegaBlock coverage is very good, the average was 91.59%, which means that the detection was very high and optimization was made on a significant amount of code. Other MegaBlock characteristics are present on Table 2.1: the average number of instructions per call is the product of the number of instructions per iteration with the average number of iterations, maximum ILP is the maximum instruction level parallelism and SW IPC is the amount of instructions per cycle achieved by software.

Table 2.1 — Detected MegaBlock characteristics [7].

	Megablock characteristics			
Kernels	Avg. Inst. Executed p/call	Max. ILP	Coverage (%)	SW IPC
count	192	2	94.9	0.857
even_ones	192	3	94.0	0.857
fibonacci	1,497	2	99.4	0.857
ham_dist	192	3	94.0	0.857
pop_cnt32	256	3	97.2	0.889
reverse	224	3	95.6	0.875
compress	138	3	89.7	0.889
divlu	155	2	90.5	0.833
expand	138	3	89.7	0.889
gcd	330	2	98.8	0.889
isqrt	96	3	84.0	0.857
maxstr	120	2	88.1	0.800
popcount3	15500	3	85.4	0.912
mpegcrc	465	4	87.6	0.934
usqrt	288	6	84.9	0.947
merge1	444	2.7	N/A	0.865
merge2	166	2.5	N/A	0.860

The speedup tests results present on Figure 2.4 show that from the 3 different architectures used. It is proved that if used the right architecture, the speedup can be very high and even reach its maximum potential which happens for LMB-FSL with some of the benchmarks used.

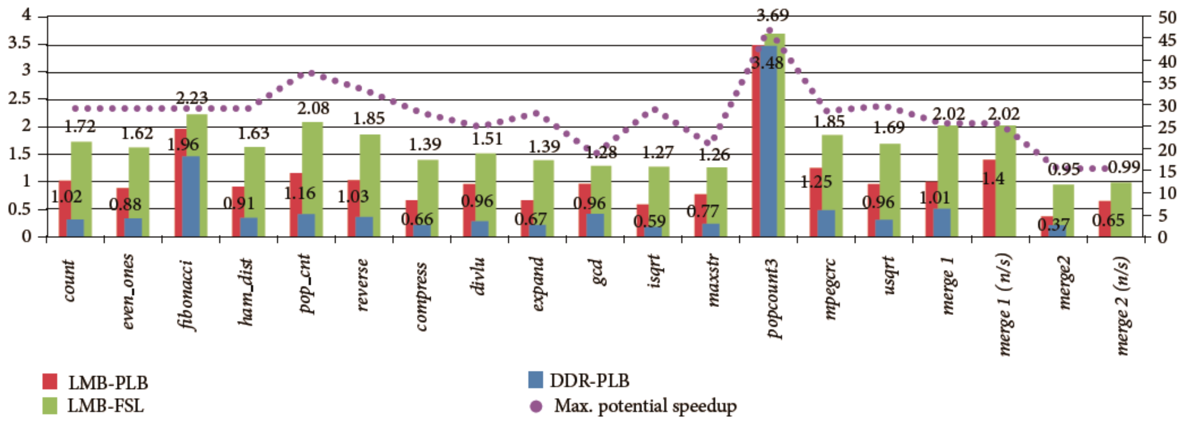


Figure 2.4 – Speedups for all three architectures. Results for DDR-PLB architecture use the axis on the right. Bar labels show the results for the LMB-PLB and LMB-FSL architectures (axis on the left). A trend can be observed for all three cases. The different overheads dictate the relative scales of the attained speedups [7].

2.1.2 - Other approaches

Nuno Paulino's work [7] included a study of other efforts on the same matter that also focus on mapping computations to RPUs during runtime but with different approaches:

- Warp processor [8, 9] is a runtime reconfigurable system which uses a custom FPGA as a hardware accelerator for a GPP. The system performs all steps at runtime, from binary decompilation to FPGA placement and routing. The running binary code is decompiled into high-level structures, which are then mapped to a custom FPGA fabric with tools developed by the authors. Warp attains good speedups for benchmarks with bit-level operations and is completely transparent. It relies on backward branches to identify small loops in the program.
- AMBER [10, 11] uses a profiler alongside a sequencer. The sequencer compares the current Program Counter (PC) with previously stored PC values. If there is a match, it configures the proposed accelerator to execute computations starting at that PC. The accelerator consists of a reconfigurable functional unit (RFU), composed by several levels of homogeneous functional units (FUs) placed in an inverted pyramid shape, with a rich interconnection scheme between the FUs. The RFU is configured whenever a basic block is executed more times than a certain threshold. Further work considered a heterogeneous RFU [10], and introduced a coarser-grained architecture to reduce the configuration overhead. The AMBER approach is intrusive as the RFU is coupled to the GPP's pipeline stages.

- CCA [5, 6] is composed of a reconfigurable array of FUs in an inverted pyramid shape, coupled to an ARM processor. The work addresses the detection of computations suitable to be mapped to a given CCA, as well as discovering a CCA architecture that best suits a set of detected control-data flow graphs (CDFGs). Initially, the detection was performed during runtime, by using the rePLay framework, which identifies large clusters of sequential instructions as atomic frames. The detection was later moved to an offline phase, during compilation [6]. Suitable CCA CDFGs are discovered by trace analysis, and the original binary is modified with custom instructions and rearranged to enable the use of the CCA at runtime.
- The DIM reconfigurable system [3, 4] proposes a reconfigurable array of FUs in a multiple-row topology and uses a dynamic binary translation mechanism. The DIM array is composed of uniform columns, each with FUs of the same type. DIM transparently maps single basic blocks from a MIPS processor to the array. DIM also introduced a speculation mechanism which enables the mapping of units composed by up to 3 basic blocks. The system is tightly coupled to the processor, having direct access to the processor's register file.

2.2 - Concept of Megablock

Previous work [12] has focused on moving sequence of instructions from a GPP to a RPU during runtime. Even though the RPU will process these instructions faster, there is a communication overhead generated by moving computations. But loops tend to execute for a longer time so, they have a higher potential for improvement, which means that by moving entire loops, the possibility of amortizing the communication overhead is very high and the outcome will most likely result into a processing boost. The Figure 2.5 exemplifies the identification of the several types of code segments that are going to be further explained next.

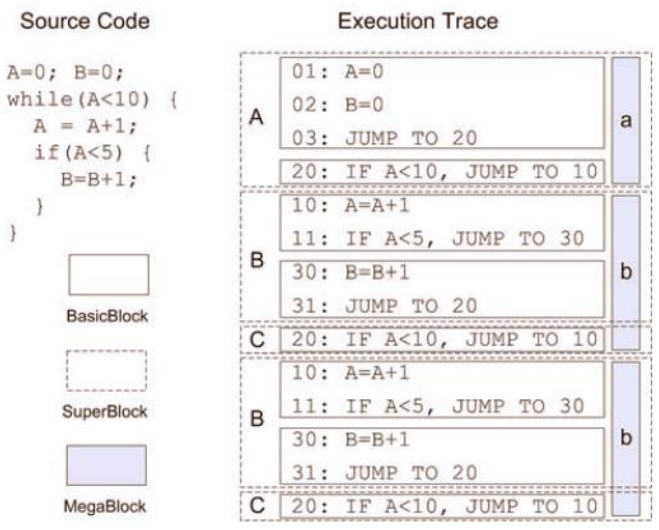


Figure 2.5 – Execution trace partitioning according to different granularities: a) BasicBlocks; b) SuperBlocks; c) MegaBlocks [13].

The detection of the code segments that are worth moving to the RPU begins by first identifying a BasicBlock. Each BasicBlock consists of a sequence of instructions with only one entry point and one exit point. So, a BasicBlock ends with the detection of a jump or branch and then, another BasicBlock starts.

The next step is to identify SuperBlocks [14]. A SuperBlock is formed by adding BasicBlocks until a backward jump is reached. The target address of the backward jump will then, start another SuperBlock, a hash value is created with this address and it allows to uniquely identify each SuperBlock. Thus, it has only one entry point but multiple exits as it can be formed by several BasicBlocks that can verify conditions to decide if the trace execution continues within the SuperBlock or exits.

Finally, Megablocks are formed by using the same concept that is used to create SuperBlocks but this time, with them as the “source”, this is, by adding SuperBlocks until they repeat themselves. When adding SuperBlocks, there has to be a predetermined maximum size for the Megablock. Tests concluded [7], that 32 is a good amount of maximum consecutive SuperBlocks as usually there are no significant gains for MegaBlocks with a bigger size. The Megablock is a sequence of instructions that tend to repeat several times and each repetition is an iteration. Even sequences of SuperBlocks that don’t repeat themselves are considered Megablocks but with only 1 iteration. The identification and extraction of MegaBlocks can easily be made at runtime by analyzing a stream of SuperBlock hashes. Results shown on Figure 2.6 prove that the usage of Megablocks with a size of up to 32 Superblocks is a better partitioning method than only using BasicBlocks, Superblocks or Warp processor’s method [8] which uses the Superblock concept.

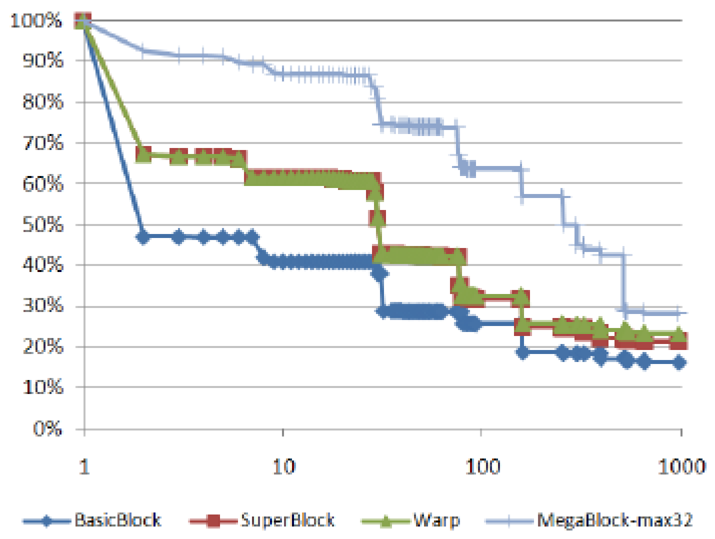
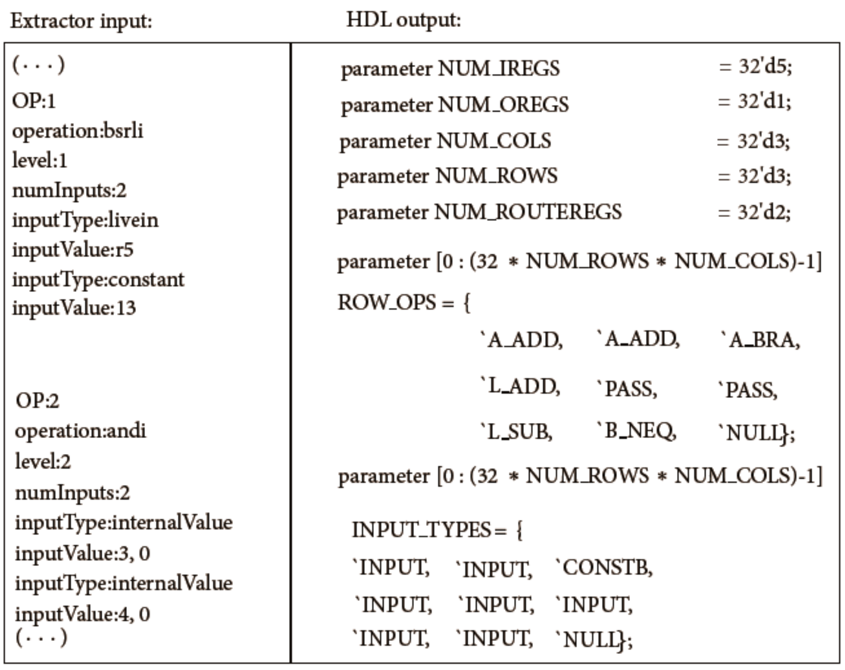


Figure 2.6 –Portions of the trace (Y axis) covered by blocks, identified by several partitioning techniques, which have at least a certain amount of iterations (X axis) [7].

2.3 - HDL Generation Tool

A toolchain has already been develop [7] with the purpose to detect Megablocks and generate an RPU and its configuration bits. The HDL description and routing generation tool parses Megablock information, determines FU sharing across Megablock graph representations, assigns FUs to rows, adds pass through units, and generates a file containing the placement of FUs and data required for Megablock identification. Only one Megablock is executed on the RPU at a time so FUs are shared between different Megablocks. Each call treats a single Megablock so, to generate a combined RPU description for several Megablocks, the tool maintains information between calls.

The Figure 2.7 shows an excerpt of a RPU HDL header file that this tool creates (on the right side) and it's respective Megablock information that serves as input (on the left side). The HDL header fully characterizes the RPU as it specifies the number of inputs/outputs and routing registers, the number of rows and columns of the RPU, the placement of functional units (FUs) and its constant value operators, and other auxiliary parameters.



Extractor input:

```
(· · ·)
OP:1
operation:bsrli
level:1
numInputs:2
inputType:livein
inputValue:r5
inputType:constant
inputValue:13


OP:2
operation:andi
level:2
numInputs:2
inputType:internalValue
inputValue:3, 0
inputType:internalValue
inputValue:4, 0
(· · ·)
```

HDL output:

```
parameter NUM_IREGS              = 32'd5;
parameter NUM_OREGS              = 32'd1;
parameter NUM_COLS               = 32'd3;
parameter NUM_ROWS               = 32'd3;
parameter NUM_ROUTEREGS          = 32'd2;
parameter [0 : (32 ∗ NUM_ROWS ∗ NUM_COLS)-1]
ROW_OPS = {
               `A_ADD,   `A_ADD,   `A_BRA,
               `L_ADD,   `PASS,    `PASS,
               `L_SUB,   `B_NEQ,   `NULL};
parameter [0 : (32 ∗ NUM_ROWS ∗ NUM_COLS)-1]
 INPUT_TYPES = {
 `INPUT,  `INPUT,  `CONSTB,
 `INPUT,  `INPUT,  `INPUT,
 `INPUT,  `INPUT,  `NULL};
```

Figure 2.7 - RPU HDL header excerpt [7].

2.4 - Interactive Application

Python was first chosen as the implementation programming language for the multipath and for the interactive tool because it's available for all major operating systems, has a simple and concise syntax, many good resources available and is simple and easy to debug (which 0increases productivity). The latest version of Python would be desirable but some of the chosen tools and libraries might not support it so this depends on the requirements of the packages that would be used.

Windows OS was used to install and test the tools because it is the most used OS, and usually it is harder to install than for example on UNIX or MAC OS so if it works on Windows, it will pretty much work on the other relevant OS. Based on Python programming, the next tools to develop the interactive application were considered:

- Matplotlib [15] is a Python 2D plotting library that allows flexible drawing of graphs. It was not considered as a tool to work on but it is a requirement for some other tools. It also requires other libraries (numpy, libpng, freetype, dateutil and pyparsing) and is available for Python 2.6, 2.7 and 3.2.
- Graphviz [16] is a graph visualization software. It takes descriptions of graphs in a simple text language and makes diagrams in useful formats like SVG and PDF. Available for Python 2.6 and 2.7.
- Pydot [17] is a python interface to Graphviz's Dot language. Allows to easily create both directed and non-directed graphs from Python. Requires Pyparsing and Graphviz. Works with Python 2.6 and 2.7.

- Xdot [18] is an interactive viewer for graphs written in Graphviz's dot language. Requires Python 2.6 or 2.7, Graphviz, PyGTK, Pycairo and PyGobject.
- Python-graph [19] provides a suitable data structure for representing graphs and a whole set of important algorithms. Requires Python 2.6, Pydot and Pyparsing.
- Networkx [20] is a Python language software package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. Requires Matplotlib, Graphviz, Pyparsing and works with Python 2.6 or later.
- NodeBox [21] is a node-based software application for generative design. Requires Python 2.5 or 2.6 and Pyglet.
- Igraph [22] is a software package to create and manipulate undirected and directed graphs. Requires Python 2.4 or later and Pycairo.

There is a python repository [23] , though unofficial, has a very big database of Python extension packages for windows and is of great help for developing on this programming language.

Though the preference on working with Python, after more thoughts on the matter, decision was made to work on java instead since it was the language used to develop the MegaBlock Extractor and allows to have easier access to data created from it. A couple of experiments done with Jgraph tool soon made it the selection for this work. It's a graph visualization library with interaction capability, Swing compatible and simple to use. It is based on the graph theory which is basically representation of nodes and edges connecting them, exactly what was needed for this work.

Previous work [12] includes a Megablock intermediate representation based on a graph structure. It uses four types of nodes and five types of connections as Figure 2.8 shows.

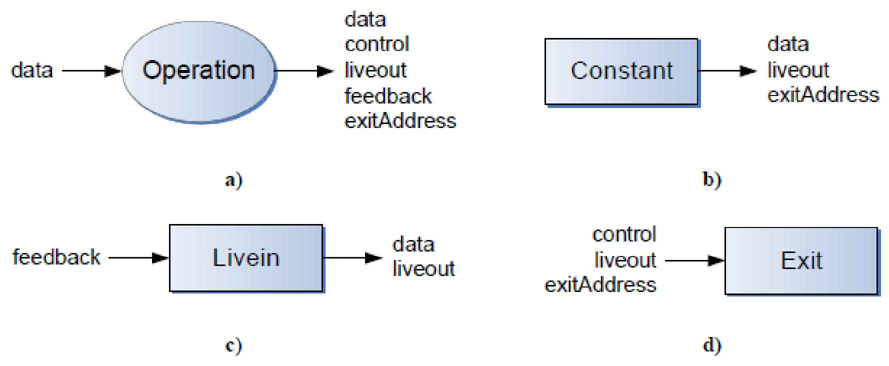


Figure 2.8 - Types of nodes and possible connections in a Megablock graph. [12].

The first node (Figure 2.8 a)), represents an operation. Figure 2.8 b) is a constant node which indicates its value and cannot be changed. *LiveIn* node on Figure 2.8 c) represents an external value. The *Exit* node (Figure 2.8 d)), represents an exit point of the Megablock.

Figure 2.8 also shows all the five different types of connections and how they can be linked to each type of node to represent all the possible interactions between them.

Data connections represent the flow of data between outputs and inputs of some nodes. It is labeled in the format "OUT:IN" where OUT is the output index of the source node while IN is the input index of the destination node. Operation, Constant and *LiveIn* nodes can all be sources of data connections but only Operation nodes can be destinations. *LiveIn* and *Exit* nodes can still receive data but it has some particularities so they are considered of another type which is explained ahead.

Control connections are boolean values which indicate if exit points are triggered or not. It always has an operation as source and an exit as destination, therefor its label is in the format "OUT" to indicate the output index of its source.

Liveout is a data connection in the format "OUT:SYSTEM_VAR" where as usual "OUT" indicates the output index of the source and "SYSTEM_VAR" indicates the system variable to be updated. This connection can only have *Exit* nodes as destinations.

Feedback is also a data connection but this one represents internal updates to values that were initially fetched. Only *LiveIn* nodes can be destinations and so, the format used for it's label is "OUT" to indicate the output index of the source operation.

exitAddress can only have *Exit* nodes as destinations and it indicates the instruction address from where the processor resumes execution for that particular Exit. Its label indicates the output index of the source node and is in the format "OUT".

Whenever a connection doesn't have an operation node as source, its label "OUT" field is simply left blank.

This concept was adopted by the jgraph graph representation used on the interactive tool, but to be in accordance with jgraph's literature, nodes will be called vertices and connections will be edges.

2.5 - Multipath Extension of Megablock Extractor

The initial approach was to develop the multipath in Python language and to read the files resulting from the Megablock Extractor processing to analyze and make the optimization. Even though with the Java approach, those files are still used to read Megablock data, this way it is possible to use some methods from the Megablock Extractor to recover the original java objects that originated that data and therefor have access to more detailed information. On a downside, this required a deep study to the Megablock Extractor to understand how it generates the data and to know how to use it while on the first approach, the data would simply be read from the files but still, this can be beneficial as some processing that Multipath needs can already be found in the Megablock Extractor.

In certain cases like when a program has two MegaBlocks that need to be executed several times, the mapper choses the one that needs to be executed the most to run on the CGRA. At a first thought this approach looks good and seems not to have any problem with it but in some cases, despite of that Megablock being executed more than the other, it doesn't mean that all its iterations are done in a row, it can be constantly interrupted by the need to run the other one. And if the difference between the number of iterations of each one is low, it means that the Megablock chosen can be interrupted a lot of times and so, mapping to the CGRA can end up not being profitable due to all the overhead generated by constantly changing execution between it and the processor. The best way to solve this is to somehow merge both Megablocks into just one so that it could work as both and load it to the CGRA.

This requires a deep view into how each Megablock operates and comparisons between them in order to find out what they have on common and what changes need to be done to make them work as one. To analyze all vertices of a graph, a Graph Traversal technique was used: by using the depth first search algorithm, all vertices are guaranteed to be visited, and by doing this simultaneously on two graphs, comparisons are made and information are gathered to then decide which parts of those graphs can be merged and how. An explanation of the Depth First Search can be found next on 2.6.1 and the algorithm that uses this can be found on chapter four where it is explained with detail.

2.6 - Graph Traversal Algorithms

A part of the Multipath requires the comparison between two Megablock's contents and their analysis to figure out which parts are mergeable or not. For this, a graph traversal algorithm was used to do the part of going through all vertices of both graphs. There are two interesting types of graph traversals: Breadth First Search examines all the connected nodes of the one it is visiting and then moves to one of them while putting the others into a queue to be visited later. The Depth First Search starts on a node and follows a random path until it finds its end and then backtracks and revisits nodes to find other paths with unvisited nodes. This is repeatedly done until all nodes are discovered. The Depth First Search approach is the one that fits Multipath algorithm better and so, it was the used one. Figure 2.9 represents it:

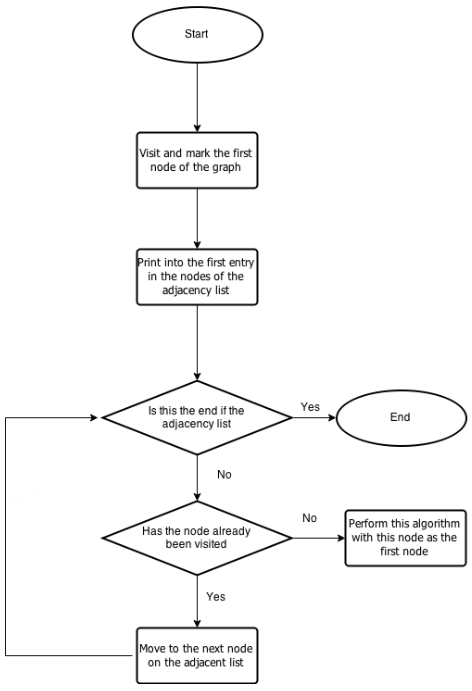

Start
Visit and mark the first node of the graph
Print into the first entry in the nodes of the adjacency list
Is this the end if the adjacency list
Yes
End
No
Has the node already been visited
No
Perform this algorithm with this node as the first node
Yes
Move to the next node on the adjacent list

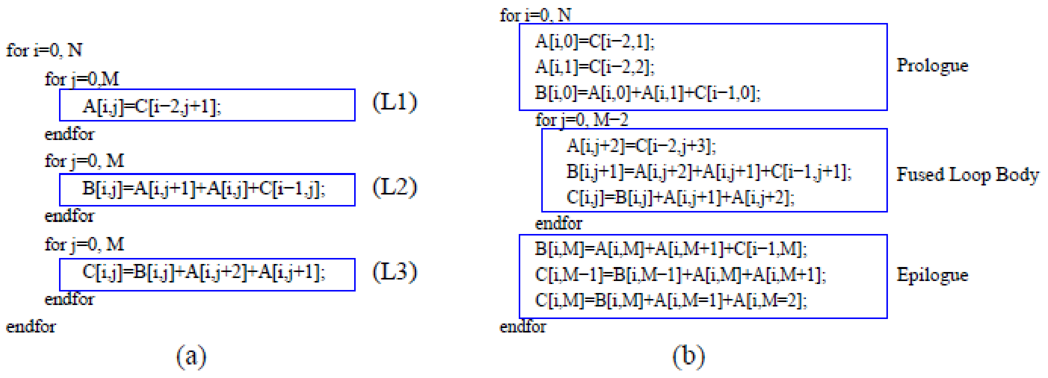
Figure 2.9 - Flowchart of the DFS traversal algorithm.

2.7 - Loop transformation techniques

Some other techniques based on loop transformations were also studied. Even though these transformations are intended to affect directly program's code, the same basis was used but with runtime information and a graph approach for the Multipath extension of the application developed on this work.

2.7.1 - Loop fusion

Loop fusion is a powerful program transformation that can improve the timing performance of both sequential and parallel programs. As the name suggests, it reduces the number of loops by merging them. But simple loop fusions aren’t always applicable because of the existence of conflict data dependences among loops. Next is shown an example of a loop fusion problem.



```
for i=0, N
    for j=0,M
        A[i,j]=C[i−2,j+1];                     (L1)
    endfor
    for j=0, M
        B[i,j]=A[i,j+1]+A[i,j]+C[i−1,j];       (L2)
    endfor
    for j=0, M
        C[i,j]=B[i,j]+A[i,j+2]+A[i,j+1];       (L3)
    endfor
endfor
                (a)
```

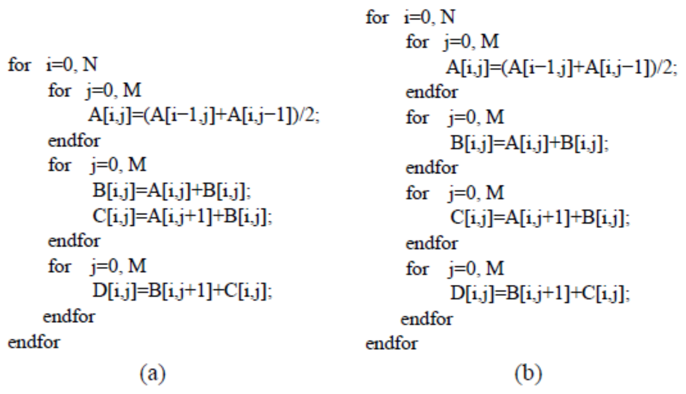
```
for i=0, N
    A[i,0]=C[i−2,1];
    A[i,1]=C[i−2,2];                           Prologue
    B[i,0]=A[i,0]+A[i,1]+C[i−1,0];
    for j=0, M−2
        A[i,j+2]=C[i−2,j+3];
        B[i,j+1]=A[i,j+2]+A[i,j+1]+C[i−1,j+1]; Fused Loop Body
        C[i,j]=B[i,j]+A[i,j+1]+A[i,j+2];
    endfor
    B[i,M]=A[i,M]+A[i,M+1]+C[i−1,M];
    C[i,M−1]=B[i,M−1]+A[i,M]+A[i,M+1];         Epilogue
    C[i,M]=B[i,M]+A[i,M=1]+A[i,M=2];
endfor
                (b)
```

Figure 2.10 - Loop fusion example with data dependency problem [25].

Figure 2.10(a) represents a program with 3 loops (L1, L2 and L3) inside another loop. Despite of all three loops having the same characteristics (they all go from j=0 till j=M), they all cannot be fused without making proper changes. If we take a closer look at the content of each loop, we will see that there are some data dependencies that wouldn't allow a simple fusion. The computation of some variables depend on a future iteration. On L2, B[i,j] depends on A[i,j+1] so, each iteration of B couldn't be calculated because it depends on a value that is only going to be available on a future iteration. The same problem occurs on L3, C[i,j] depends on A[i,j+1] and A[i,j+2] which also creates another dependency problem with L2 as it depends on C[i+1,j]. To solve these fusion-preventing dependency problems, loop shifting or retiming can be used. A prologue and an epilogue have to be added when creating a fused loop as shown on Figure 2.10(b). By using this technique, the loop decision overhead is reduced as we will only have 1 loop instead of 3 but it creates a longer sequence of instructions and if used on low iteration loops, it could not be profitable. But MegaBlocks tend to have high iterations so by applying it on this dissertation, it is expected to have good results.

2.7.2 - Loop Distribution with Direct Loop fusion

Loop distribution is a technique that works on the reverse way of loop fusion. This technique is usually used on large loops that don't fit into cache. The point is to separate some instructions inside the same loop, into multiple loops. Figure 2.11(a) shows a code example with three inner loops and one outer loop, Figure 2.11(b) shows the result of this technique.

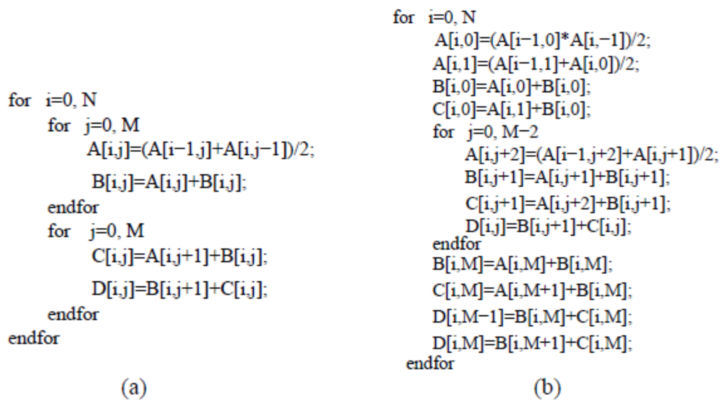


```
for  i=0, N
     for  j=0, M
          A[i,j]=(A[i−1,j]+A[i,j−1])/2;
     endfor
     for  j=0, M
          B[i,j]=A[i,j]+B[i,j];
          C[i,j]=A[i,j+1]+B[i,j];
     endfor
     for  j=0, M
          D[i,j]=B[i,j+1]+C[i,j];
     endfor
endfor
               (a)
```

```
for  i=0, N
     for  j=0, M
          A[i,j]=(A[i−1,j]+A[i,j−1])/2;
     endfor
     for  j=0, M
          B[i,j]=A[i,j]+B[i,j];
     endfor
     for  j=0, M
          C[i,j]=A[i,j+1]+B[i,j];
     endfor
     for  j=0, M
          D[i,j]=B[i,j+1]+C[i,j];
     endfor
endfor
               (b)
```

Figure 2.11 - Loop distribution example [25].

On the original code, there was a loop that accessed two arrays (B[i,j] and C[i,j]) while after loop distribution, each array is accessed on a separate loops and the processor only needs to access 1 array at a time. This might look like counter-productive on a speed optimization point of view because it adds 1 more loop and so, the decision overhead is increased. But by applying Loop Distribution with direct loop fusion, the result is a mid-term between speed optimization and code size as it can be seen on Figure 2.12(a).



```
for  i=0, N
     for  j=0, M
          A[i,j]=(A[i−1,j]+A[i,j−1])/2;
          B[i,j]=A[i,j]+B[i,j];
     endfor
     for  j=0, M
          C[i,j]=A[i,j+1]+B[i,j];
          D[i,j]=B[i,j+1]+C[i,j];
     endfor
endfor
               (a)
```

```
for  i=0, N
     A[i,0]=(A[i−1,0]*A[i,−1])/2;
     A[i,1]=(A[i−1,1]+A[i,0])/2;
     B[i,0]=A[i,0]+B[i,0];
     C[i,0]=A[i,1]+B[i,0];
     for  j=0, M−2
          A[i,j+2]=(A[i−1,j+2]+A[i,j+1])/2;
          B[i,j+1]=A[i,j+1]+B[i,j+1];
          C[i,j+1]=A[i,j+2]+B[i,j+1];
          D[i,j]=B[i,j+1]+C[i,j];
     endfor
     B[i,M]=A[i,M]+B[i,M];
     C[i,M]=A[i,M+1]+B[i,M];
     D[i,M−1]=B[i,M]+C[i,M];
     D[i,M]=B[i,M+1]+C[i,M];
endfor
               (b)
```

Figure 2.12 - Code results after applying a) Loop distribution with direct loop fusion and b) Loop fusion [25].

By comparing with the Loop fusion technique on Figure 2.12, Loop Distribution with direct loop fusion still has more loops but it also has a much reduced code size. This technique allows a reduction on the number of loops while still maintaining a low sized code, so it can be used if optimization with the loop fusion technique is problematic by resulting into a very big code size.

This chapter described all the research done before starting developing the application. It explains its target architecture developed by Nuno Paulino [7] and the concept of Megablock. It also explains the choices for the technologies used on the implementation of the application. The approach used on the Multipath and several techniques related to it were also discussed. The following chapter is about the most important components of the application and how they work together. It explains how the organization of this project was done.

Chapter 3

Architecture of the Interactive Megablock Merging Tool

As 1.2 describes, the main objectives of this dissertation are to develop an interactive application and a Multipath extension for the Megablock Extractor. Which means that the project would have two main streams and so, it was clear that to achieve a good organization, two packages should be created, one for each objective. "`GUI`" which contains all the classes that deal with visualization and interactions such as representation of Megablocks, Operations and menus. And "`MegaBlockMultipath`" that has the data processing classes responsible to read information from the files generated by the Megablock Extractor Tool to create objects from the GUI Package in order to present a correct representation of all Megablocks and to make all the necessary changes when a merge is requested.

Due to the high complexity of the application developed and to the high amount of components used to implement the solution, only the most significant ones are described in this section. Auxiliary methods and variables are sometimes mentioned but their explanation isn't done because they just do a simple task and their names are most of the time self-explanatory. The next figures (3.1 and 3.2) show the organization of the work done on this dissertation as the main components. For detailed view on the full architecture, Appendixes section can be consulted as it has figures of all components of each class used.

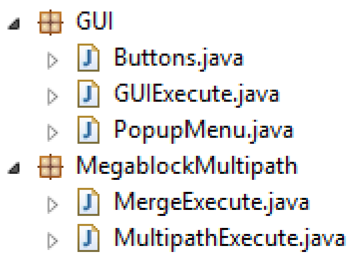


Figure 3.1 - Project's file organization

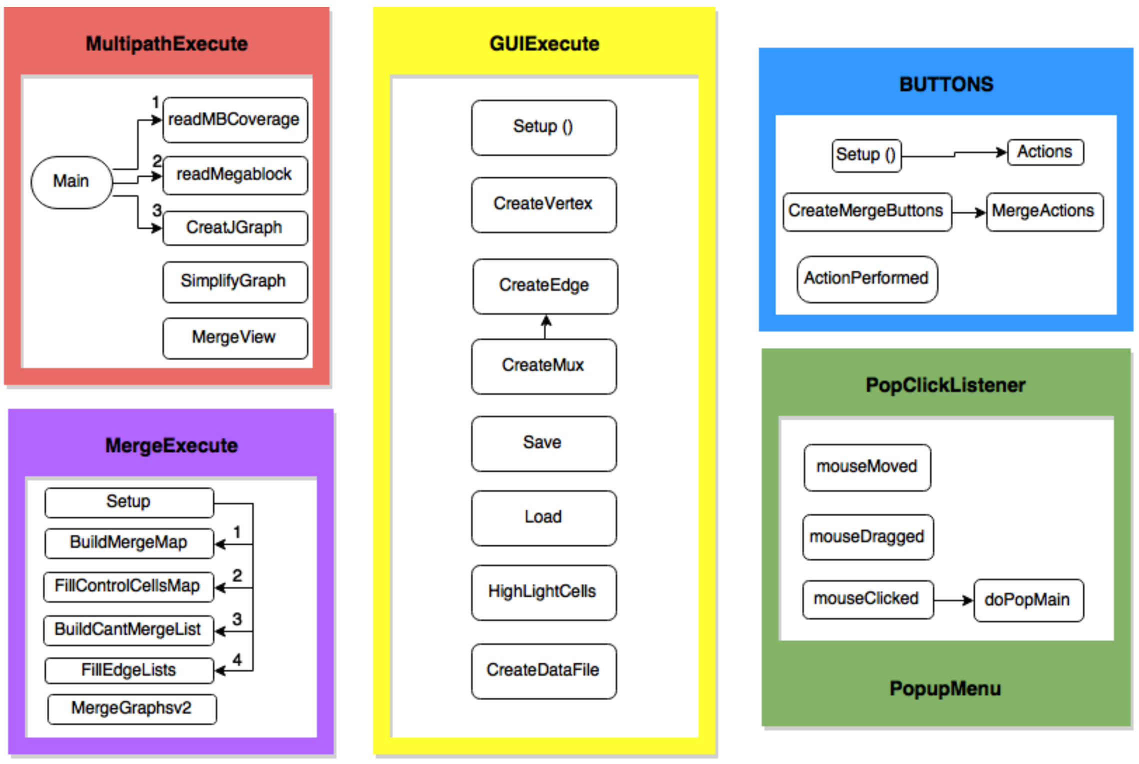


Figure 3.2 - Overview of all Classes of the Project

3.1 - Organization of the MegaBlockMultipath Package

This project has mainly two kinds (types) of processing information and therefore, two java files were created, each one with its own purpose. The application has a need to read information from the Megablock Extractor and to create objects that would visually represent them, so, "`MultipathExecute.java`" was created and it acts mostly like a link between the MegaBlock Extractor and the jgraph visualization system used which is organized into the "`GUI`" package. The other file created is "`MergeExecute.java`" and it contains more complex data manipulation relative to merging graphs. It focuses on comparing jgraphs already created and it allows creating a different one (merged graph) that represents two Megablocks at once.

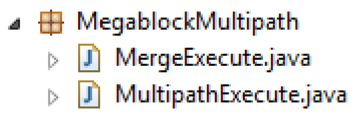


Figure 3.3 – MegablockMultipath pack organization.

3.1.1 - The MultipathExecute Class

This is the main class of the project. It reads the files extracted from Megablock Extractor, it creates the main graph, all the graphs that represent each Megablock and the simplified versions as well. When the interactive tool is launched, the main class of this file is executed. It uses "`readMBCoverage`" to get the "`coverage`" file from the MegaBlock Extractor directory, which provides global information about the program executed. Then it uses "`readMegablock`" to read all information of each MegaBlock detected and it ends by using the "`CreateJGraph`" method to build the main graph and all MegaBlock graphs. "`readMBCoverage`" and "`readMegablock`" use SPECs methods to retrieve information from the files while "`CreateJGraph`" uses "`GUIExecute`" to create all the graphs. After this is done, the application just responds to interactions from the user. "`SimplifyGraph`" is activated when the "simplify" button is pressed and it uses the same "`GUIExecute`" methods than "`CreateJGraph`" to create a simplified graph. "`MergeView`" is activated by the button with the same name on the popup menu. It shows both graphs side-by-side and it uses "`MergeExecute`" for comparisons between each `Megablock's` content and it uses "`Buttons`" to change the top menu accordingly.

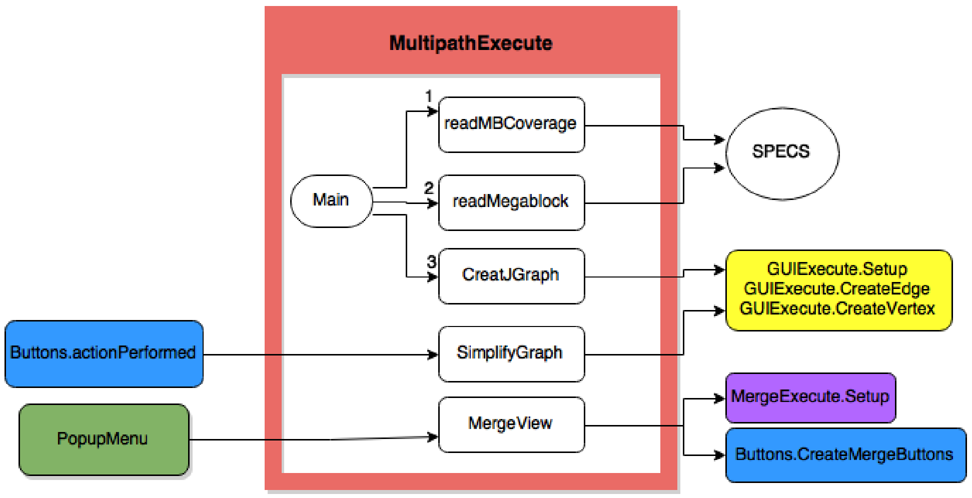


Figure 3.4 - Organization of the MultipathExecute content.

Table 3.1 — Resumes of the main MultipathExecute components.

Component	Function
readMBCoverage	Reads global information from the “Megablock Extractor”
readMegablock	Reads information related to a specific “MegaBlock” from “Megablock Extractor”
CreateJgraph	Creates a graph
SimplifyGraph	Creates a simplified version of a graph
MergeView	Shows two mergeable graphs in a side-by-side view

- **`readMBCoverage`**: searches into the output directory of the MegaBlock Extractor for the “`(…)mbcoverage.xml`” file and with aid of some SPECS methods, a “coverage” object is retrieved from it. This object contains several information about the Megablocks extracted and it is used to build the “main graph” by creating a representation of each Megablock with the `GUIExecute.createvertex` method and ordering them according to their memory location by connecting them with edges, using the “`GUIExecute.createedge`” method.

- **`readMegablock`:** creates for each vertex, on the main graph, another graph with the content of its respective MegaBlock. This is done by “`readMegablock`” which uses SPECS methods to get a “`MicroBlazeGraph`” object from the “(…)`microblaze-megablock.xml`” file that corresponds to the current MegaBlock that is going to be created. This method is used as many times as the amount of Megablocks so that each one is then created on “`CreateJGraph`”.

- **`CreateJGraph`**: uses the object gotten from “`readMegablock`” to obtain information about the nodes and connections regarding the instructions that are part of the Megablock and to create vertices and edges on the graph. When these cells are created, they receive the same attributes as the nodes and connections have. Each *Operation, Constant*, *LiveIn* and *Exit* has a unique ID in the format of its type followed by a number e.g. “constant_X” for a constant, where X is a different number for each constant. They also have its unique label but each type of node has its own format for it. *Constants* have their value as label, *LiveIns* show their register followed by “(input)” e.g. “r6(input)” for the variable r6, *Exits* labels have similar format as

their IDs which is "Exit:X" where X is again a different number for each *exit* vertex and *operations* use the format "X:OP" where X is a different number and OP is the operation type of the vertex e.g "4:add" for an add vertex. There are two more attributes (address and instruction) but these are exclusive to *operations* due to their nature. The address indicates directly an operation location in the memory and the instruction is the assembly instruction of the operation. From the instruction it is possible to obtain the operation's inputs and outputs though these are also represented on the edges connected to it.
When creating the jgraph, there are some particularities to pay attention. For some reason, the Megablock Extractor duplicates some constants, so when they are detected, "`CreateJgraph`" method insures that they aren't added again and that the only constant vertex created has all the edges that were connected to both constant nodes. Some operations have different IDs and labels but can have same address and instruction. This happens when the exactly same instruction repeats itself but gets its inputs/output from different sources. Due to Megablock's repetitive nature, this happens quit often, the same operations are executed many times and the presence of conditions makes it to receive and send data to different vertices. So in that case, the operation that represents that instruction needs to be connected to different vertices and so, another vertex is created representing the exact same instruction. There is also a special case regarding the store and load operations. These operations are interpreted by the MegaBlock Extractor as two each. An "addi" that adds an offset to the register in order to correctly indicate the data, and the store/load operation itself. But when the store/load operation executes directly on the register without no offset, the MegaBlock Extractor doesn't create an add node and just considers the store/load operation. The problem is that when the two nodes are created from one of these operations, they both have the same address which can lead to some problems when identifying operations by addresses. So to avoid it, at the point of creation of any store and load vertices, it is added the value of 1 to its address. Since assembly addresses are all even, these will be odd and there won't be any chance to have another operation with the same address. This is reverted when the parsablegraph file is created so that it has no visible effects afterwards.

- **`SimplifyGraph`:** creates the simplified version of a graph. When the "simplify" button is selected in the upper menu of a graph, this creates another representation of it with only operation vertices and the edges between them. It uses the information about the complete graph to create another "`GUIExecute`" object that only contains information about the cells that are to be represented in this simplified version.

- **MergeView**: is an option available from the popup menu of the main graph and so, it is used by "PopupMenu" class. When the user selects the Merge View option on the popup menu, the graphs relative to both Megablocks are shown side-by-side and "Buttons.CreateMergeButtons" is used to change the buttons on the top panel of each one according to this new view. "MergeExecute.Setup" is called to compare both graphs and to detect merge possibilities that are shown on this view.

3.1.1 - The MergeExecute Class

This is the class used to merge graphs. When an instance is created, the two "GUIExecute" objects corresponding to the graphs to merge have to be passed to it. "Setup" is used at the moment of creation of a "MergeExecute" object and it uses some inner methods to detect and to keep information about matching cells from both graphs as well as cells that can't be matched, control cells that will generate the control bit for the multiplexers, and also edges where those multiplexers will need to be inserted. Here is where the merge algorithm is used which is explained with detail on chapter four. This is a preparation for the merge step. When a merge is requested, "MergeGraphsv2" is called and it uses all information gathered to merge the two MegaBlocks. This can be done either from an option on the popup menu or from a button on the top panel and it is the only class that uses "GUIExecute.CreateMux".

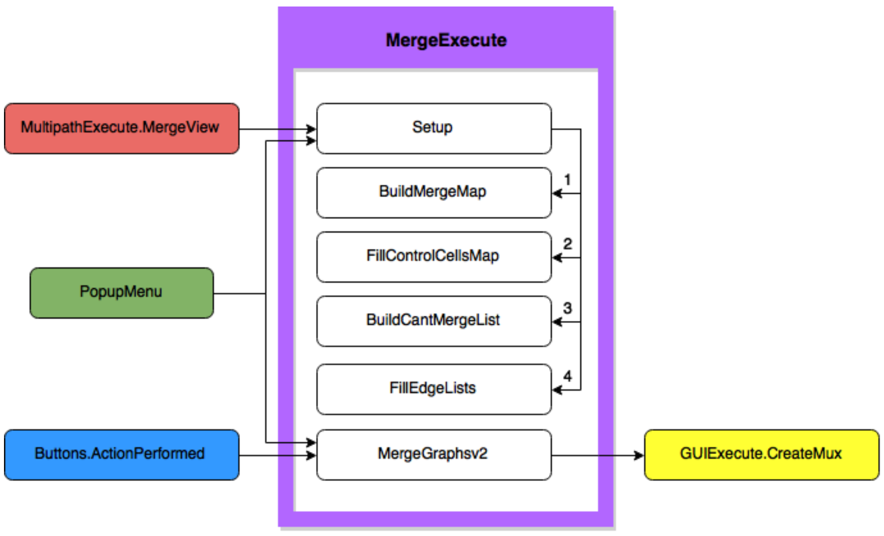


Figure 3.5 - Organization of the MergeExecute content.

Table 3.2 — Resumes of the main MergeExecute components.

Component	Function
Setup	Calls inner classes to gather all information needed for the merge
BuildMergeMap	Stores information about cells that can be merged.
FillControlCellsMap	Stores information about the control cells.
BuildCantMergeList	Stores information about the cells that can't be merged.
FillEdgeLists	Stores information about the location where multiplexes are going to be inserted.
MergeGraphsv2	Merges two graphs.

- **`Setup`:** uses "`BuildMergeMap`", "`FillControlCellsMap`", "`BuildCantMergeList`" and "`FillEdgeList`" in the correct order and with the right parameters to build all the lists and maps that are needed on a possible merge.

- **`BuildMergeMap`**: uses a recursive method: "`HelpFillMergeMap`" to analyze cells from both graphs and to make associations between them. It also uses the "`CompatibilityFactor`" method to compare some operations and to help making better decisions.

- **`FillControlCellsMap`**: uses the instruction that makes the decision on which Megablock to execute. It detects the corresponding vertices on both graphs and it stores that information. These cells are the ones where the bit that makes the decision of the multiplexers comes from.

- **`BuildCantMergeList`**: stores information about cells from one graph that aren't corresponded on the other one. It is used to make some vertices to only activate according to the control bit of the multiplexers.

- **`FillEdgeList`***:* marks the edges that connect a vertex that is on the "merge list" to one that is on the "can't merge list". This represents edges that need to have multiplexers so that the vertices on the "can't merge list" are only active when they belong to the Megablock that is being executed at the time.

- **MergeGraphsv2** is the method called to start the merge process. It can be activated from "Buttons" or from "PopupMenu". It checks all the information gathered from: "BuildMergeMap", "FillControlCellsMap", "BuildCantMergeList" and "BuildEdgeList" when "MergeExecute" was created and it uses methods from "GUIExecute" to make the appropriated changes on the Megablock graph that is about to become the merged one, such as removal and creation of vertices and edges, and multiplexers as well. It also operates on the main graph to replace both MegaBlock representations by the merged one.

3.2 - Organization of the GUI Package

This package contains all the visualization type of objects and therefore files were created to separate the main ones. GUIExecute is the main file and it contains all cells such as vertices and edges of a graph while the other files contain objects that are used to create elements to insert into the GUIExecute object and to complement it. Buttons creates a panel shown on the top of each graph with some options to help the user to interact. PopupMenu creates a specific menu that pops up on the main graph and also listens and handles mouse actions and movements done on the GUIExecute object.

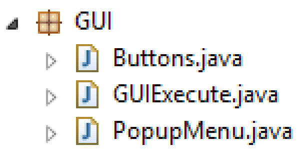


Figure 3.6 - Organization of the GUI package.

3.2.1 - The GUIExecute Class

This class creates a jgraph object and also stores all the data associated to it. It contains several lists and hashmaps to group cells of the same type and to associate their characteristics. It also has a set of methods that work around those hashmaps and lists in order to allow retrieving a cell object given any type of attribute or the other way around and even get an attribute by providing another, e.g it is possible to get a vertex ID by passing its label to the correct method. GUIExecute also allows to make changes on its graphs content e.g. create/remove cells and change their color or label. It also provides methods that retrieve a cell connected to other on any other way, e.g it enables the possibility to get an edge source or destination vertex. These methods that can detect, which cells are connected to which and how they are connected, are available from the jgraph library. GUIExecute class has so many elements that use each other's in so many ways and their names also speak for themselves. However it is necessary to explain the methods that stand out.

`GUIExecute` basically holds and provides all information about the graph and the nodes that originated it and it has methods that allow other classes to make changes on the graph. When it is created, it also creates a `Buttons` and a `PopupMenu` class associated to it. Details about them can be found on the next two sections of this report.

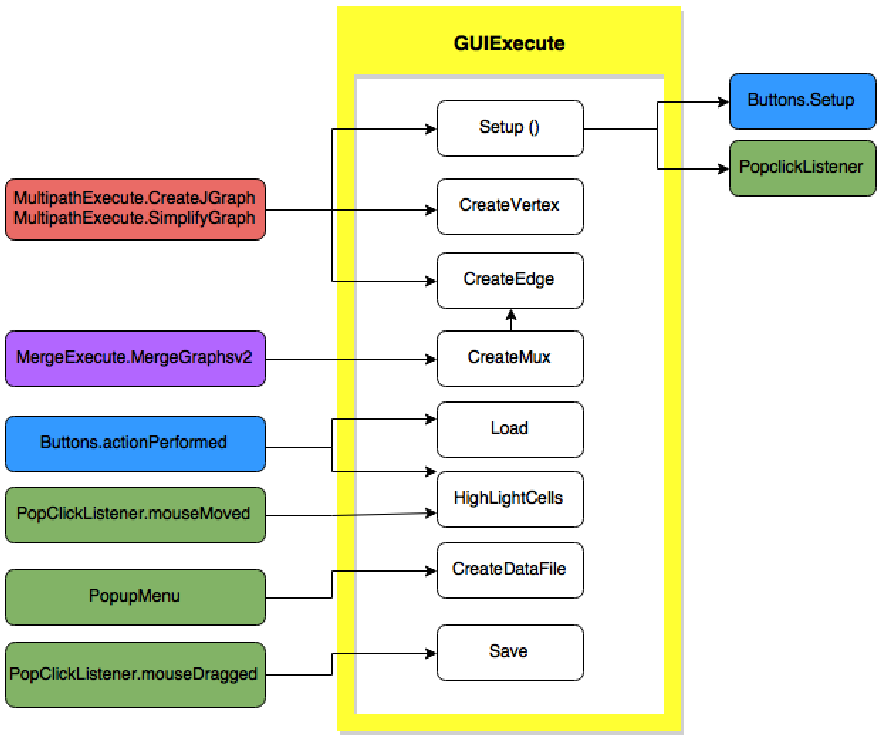


Figure 3.7 - Organization of the GUIExecute content.

Table 3.3 — Resumes of the main GUIExecute components.

Component	Function
Setup	Creates an empty graph.
CreateVertex	Creates a vertex.
CreateEdge	Creates an edge
CreateMux	Creates a mux
Save	Saves all information of the graph
Load	Loads information of a graph
HighLightCells	Highlights selected cells
CreateDataFile	Creates a data file

- **Setup:** is used whenever a graph representation needs to be created. It creates a jgraph instance and uses Buttons.Setup and PopClickListener to create and associate a button panel and listener for the popup menu. Since Setup builds an empty graph, CreateVertex and CreateEdge need to be used by the method that calls Setup to fill up the graph with the right vertices and edges. This method is used by MultipathExecute.CreateJGraph to create a jgraph of each Megablock and it is used by MultipathExecute.SimplifyGraph to create a simplified version of a Megablock when the Simplify button from the top panel is selected. Note that it is also used by MultipathExecute.main to create the main graph but since this is only used once, it isn't represented on Figure 3.7.

- **CreateVertex:** is used by MultiPathExecute.CreateJGraph and MultipathExecute.SimplifyGraph to fill a graph with vertices according to the intended representation. As described on section 2.4, there are four possible Node types: Operation, Constant, Livein and Exit. The way they are converted into vertices is explained with detail on CreateJGraph in 3.1.1 section. CreateVertex method insures that each vertex has its color and shape according to the type that they belong to and it also builds lists to keep information about each vertex's characteristics organized. After this method is used to build all vertices of the graph, they need to be connected by edges and for that purpose the CreateEdge method is called by the same methods that call CreateVertex.

- **CreateEdge**: is used after CreateVertex, when all the vertices of a graph have been made. It is used by MultiPathExecute.CreateJGraph and by MultipathExecute.SimplifyGraph to build all the edges that connect those vertices.

- **Save**: is used to save the state of a graph once, after its creation, and whenever changes are made to its cells. PopClickListener uses Save when it detects the movement of a cell so that the "Undo" button from the Buttons panel can use the Load method to get the graph back to a previous state. The save function was implemented by converting the model of the jgraph into an xml file to store information about each cell in the jgraph system while all the data behind the graph was stored into another xml file.

- **Load**: is used by Buttons.ActionPerformed method when the "Undo" button is selected. It loads the last xml files saved by the Save method to return the graph to its previous state.

- **`HighLightCells`**: is used to highlights cells for a better understanding of the graph. It is used by `PopClickListener` when the mouse cursor hovers over a vertex to highlight it. In case of a merge-view, it also highlights the other mergeable cells related to it and this same option can be obtained from `Buttons.actionPerformed` when selecting the "merge" button related to it.

- **`CreateDataFile`**: this method is used by `PopupMenu` class when the "Create Data File" option from the popup menu is selected. It builds an xml file with information regarding the selected MegaBlock the same way that the MegaBlock Extractor does when it finishes its work. This method also creates a parsablegraph file which is the format that Nuno Paulino needs to use on his work.

3.2.2 - The Buttons Class

Creates a jpanel located on the upper side of the graph. When a graph is created, a `buttons` instance is also created and associated to it. The `Setup` is responsible to build the panel with the default buttons which are stored in the `Actions` variable, to resize it and to set its correct position in the graph window. `CreateMergeButtons` uses `MergeActions` to add buttons to the panel if needed. `ActionPerformed` is used whenever a button of the panel is pressed and it guarantees that each button does what it's supposed to. The most important components of the Buttons class are described below:

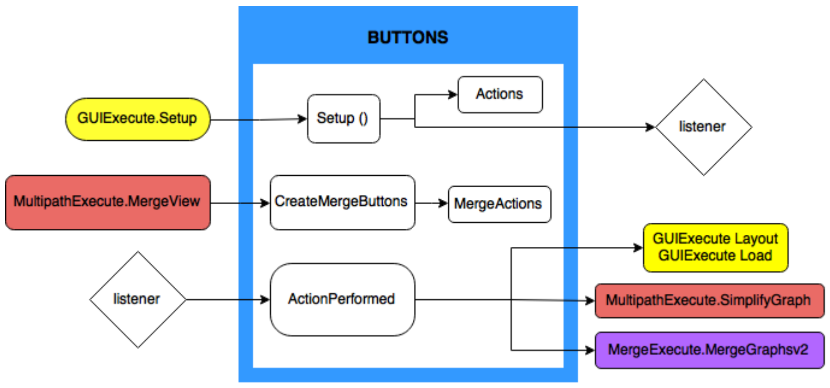


Figure 3.8 - Organization of the Buttons content

Table 3.4 — Resumes of the main Buttons components.

Component	Function
Setup	Creates the panel and inserts it on the graph
Actions	Contains the default buttons
MergeActions	Contains the merge-specific buttons
actionPerformed	Does the action according to the button pressed
CreateMergeButtons	Adds merge-specific buttons to the panel

- **`Setup`:** is used by `GUIExecute.Setup` in the moment a graph is created to build a Buttons instance and to associate it to the graph. It uses `javax.Swing` to create a JPanel on top of the graph window with all the default buttons which are stored on the `Actions` enum type of variable. It also creates a listener to detect buttons pressed and `actionPerformed` executes the correct action.

- **`actionPerformed`:** is activated when a button is pressed. By consulting the listener's information, it knows which button was selected and therefore executes the code according to that button. Detailed explanation on how each feature was implemented can be found in the `Actions` and `MergeActions` sections bellow.

- **`Actions`:** is an enum type that contains the default actions/buttons: Zoom in, Zoom out, Undo, Simplified and Reset. The next figure shows the panel created with only the default buttons followed by an explanation of each one.

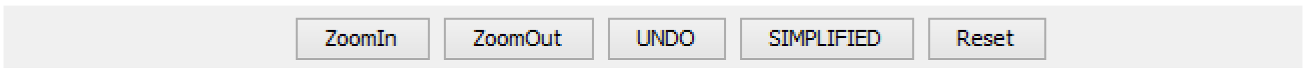


Figure 3.9 - Default buttons panel.

  - ***Zoom in***: zooms in the scale of the screen by a default amount. This feature is obtained by using the jgraph methods available for this specific purposes.
  - ***Zoom out***: zooms out the scale of the screen by a default amount. This feature is obtained by using the jgraph methods available for this specific purposes.

- ***Undo***: undoes the last change to the graph. By pressing this button, the Load method of `GUIExecute` is used to load the last save file representing the previous state of the graph.
- ***Simplified***: opens another window with a simplified version of the graph which doesn't include inputs and exits, therefore, only operations and the edges between them are represented. The `SimplifyGraph` method on `MultipathExecute` is used to create this simpler version of the current graph.
- ***Reset***: positions all the cells back to the original Hierarchical organization that is first shown when the current graph window was opened. Uses the `Layout` method on `GUIExecute` which rearranges all the cells again.

- **`MergeActions`:** If the graph is being visualized on a merge view, then some more buttons are added: the "*MergeGraphs*" button and some "Merge" ones according to the amount of groups of cells that can be merged. Next figure shows the panel with these extra buttons.

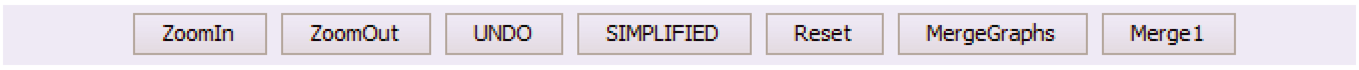


Figure 3.10 – Merge Button Panel

- ***MergeGraphs***: opens a new window with the merged graph. This button is only available on the "merge view" so a `MergeExecute` instance has already been created and all this button does is call the `MergeExecute.MergeGraphsV2` method to merge both graphs.
- ***Merge:*** by checking the amount of corresponding groups of cells between both graphs on the `MergeExecute` instance, the same amount of Merge buttons are created and when one of them is pressed, it calls the `HighlightCells` method on both graphs while passing to them the group of cells corresponding to the button clicked so that they are highlighted.

- **`CreateMergeButtons`:** is the method responsible to add the Merge relative buttons. It is used when a "MergeGraph" is created to add the Merge Buttons to the default buttons panel.

3.2.3 - Handling Popup Menus

The PopupMenu.java file has two main classes: `PopupMenu` itself which contains the model of the menu and the handler to deal with all the selections made within it, and `PopclickListener` which is responsible to handle actions made within the graph area such as mouse movements, clicks and cell movements. It also makes the popup menu appear when and where it is supposed to.

Though the popup menu is only available on the main graph, the other instances of `GUIExecute` that represent Megablocks contents also use the `PopclickListener`. This class not only creates the popup menu but it also uses listeners and handlers to keep track of the activity done with the mouse and to respond properly to each action. When created on the main graph, it makes the popup menu appear with a right click on a vertex and then, the `PopupMenu` object executes the action associated to each button clicked on the menu. When created on a Megablock graph, it keeps track of the cells that are hovered with the mouse cursor to highlight and to change colors of cells for a better interpretation of the graph by the user, it also keeps track on changes made on the layout so that it can be undone by the "Undo" button when it is selected, it also shows a vertex detailed information when a double click is done on one.

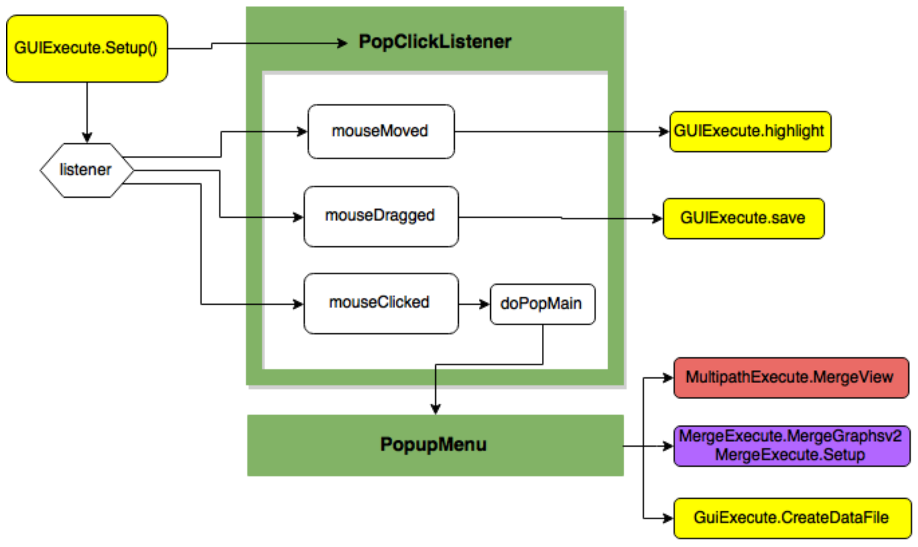


Figure 3.11 - Organization of the PopClickListener content

Table 3.5 — Resumes of the PopupMenu components.

Component	Function
mouseMoved	Highlights cells
mouseDragged	Saves graph's layout when a change is made
mouseClicked	Shows vertex's info or a popup menu
doPopMain	Creates a popup menu
PopupMenu	The menu itself and does the action according to the button pressed
PopClickListener	Does the action according to mouse inputs

- **PopclickListener**: contains the methods that are going to handle interactions with the mouse and doPopMain method to build a PopupMenu instance on the main graph. An instance of this class is created at the moment of creation of each graph so it is called by GUIExecute.Setup and it is responsible to handle the actions done with the mouse on the zone of the graph.

- **mouseMoved**: keeps reading the mouse cursor's location and uses GUIExecute.highlight to highlight the current cell in case of the listener not belonging to a merge graph. In case it is a merge graph then it highlights all the cells that belong to the mergeable tree that the current cell belongs. And in case of a merge view then it also highlights the corresponding mergeable tree of cells on the other graph.

- **mouseDragged**: checks if a cell is moved and if so, it calls GUIExecute.Save to save the graph's layout so that it can be undone if needed. When the "Undo" button from the top panel is selected, it loads the last saved state.

- **mouseClicked**: is used to detect double-clicks and right-clicks and to handle them. Whenever a double click is done in the main graph, it opens the corresponding graph window. If it is done on a vertex of a Megablock graph, then it opens a window showing detailed information about it such as its address, corresponding instruction and parent and child vertices. All this information is obtained from GUIExecute since it stores them all at the creation time of each graph. As for the right click: when it is done on a vertex, mouseClick checks if the current graph is the main graph and if so, it calls up doPopMain to make the popup menu appear with all the appropriate buttons.

- **doPopMain**: uses PopupMenu to create the menu and to adjust it according to the cell which it is related to. Because the popup menu only makes sense if the current instance of this class is related to the main graph, this method only creates it once and for that specific graph only.

- **PopupMenu**: builds the popup menu as shown on figure 3.11 and it creates inner listeners and handlers to detect selections in the menu and submenu and to make the according action take place.

Following is a figure showing this menu and explanations about each action integrated into it.

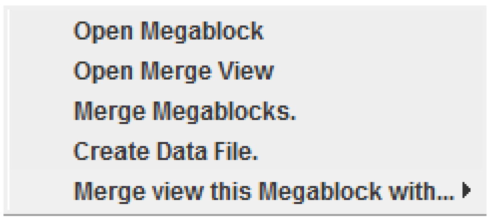
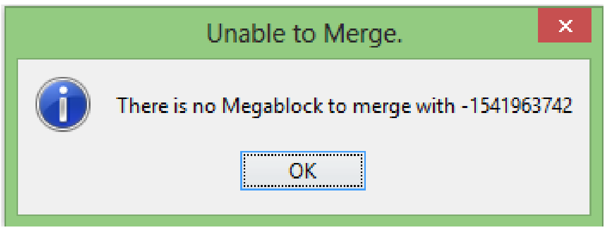


Figure 3.12 - Popup menu example.

- *Open Megablock*: this button opens the graph that shows the content of the Megablock which this menu corresponds. To do so, it uses the `SetVisible` method on the `GuiExecute` instance that has the information of this MegaBlock.
- *Open Merge View:* Due to Megablock's nature described on 3.2, the ones that start with the same address are good candidates for a possible merge. So, when this button is selected, `MultipathExecute.MergeView` is called to show the graph corresponding to the present MegaBlock and the other one with the same address on a merge view allowing the user to analyze both with further detail, compare them and to eventually perform a merge if desirable. If there is no other MegaBlock with the same starting address, the selection of this operation shows a warning message like the one bellow:

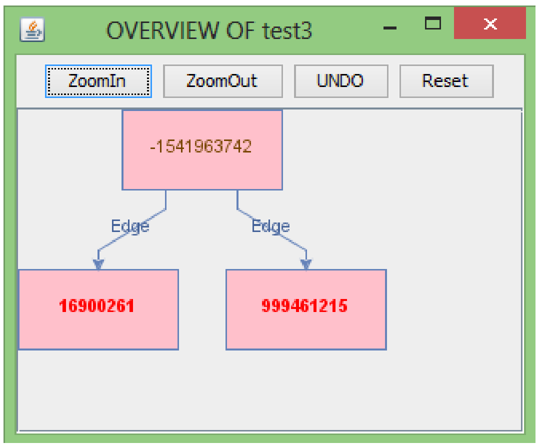

Unable to Merge.

There is no Megablock to merge with -1541963742

Figure 3.13 - Unable to merge warning message.

But despite the error, it is still possible to take a look at the merge view" of the selected Megablock with another one by using the "*Merge this Megablock with…*" button.

- ***Merge this Megablock with…***: By selecting this option, a submenu opens for the user to select which MegaBlock is intended to merge with the current one. Just like the "Open Merge View" button, this one also uses the `MultipathExecute.MergeView` method to show both graphs on the merge view. This allows the user to check for another possible merge between Megablocks that don't start with the same address.
- ***Merge Megablocks***: merges the Megablock selected with the other one that has the same address. It uses the `MergeExecute.Setup` to compare and prepare graphs for merge and then the `MergeExecute.MergeGraphsV2` to effectively do the merge. Details on how the merge is done can be found on Chapter 4. The next figure shows the main graph before and after a merge.



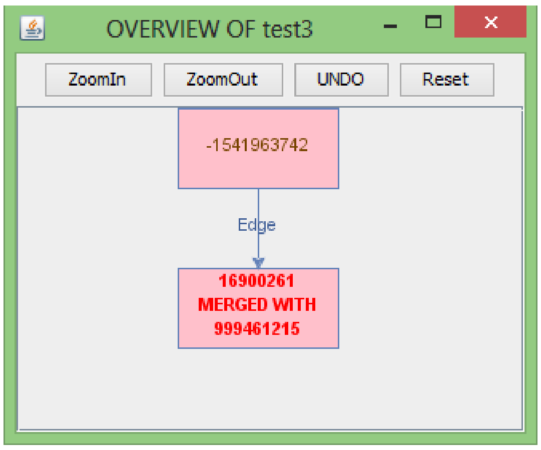


Figure 3.14 - Main graph before and after a merge.

- ***Create Data File:*** This method uses `GUIExecute.CreateDataFile` to create data files relative to the graph chosen and updated with all the changes done in it.

This chapter presented the application's structure while explaining the main parts of it. It was organized with all the classes directly related with the visual part into the `GUI` package while `MegablockMultipath` contains the ones that do most of the data related methods. It also explained with detail the function of the most important classes and methods, and how they are related. Now that the structure and organization was explained, next chapter will focus on giving a deeper insight of the solution developed to merge Megablocks.

Chapter 4

Algorithm for Megablock Merging

The diagram on Figure 4.1 represents the algorithm used to analyze both graphs and to match cells to be merged. This code is in the `MergeExecute` class and it is requested by the `PopupMenu` class when merging 2 graphs.

The technique used to analyze both graphs and to look for matches begins with the constants or registers cells since they are the inputs and calculate a “compatibility factor” between their children. These actions are used recursively until an “exit” is reached and the information is used to determinate which cells can be merged.

Inputs were used as starting points because they are the most easy to compare since they only have as attributes the “ID” and “LABEL” that must be the same. This also happens with the “Exit” cells so the technique could be used by starting on these ones and going backwards until the “inputs” are reached. In this case, the parents would have been followed instead of the children.

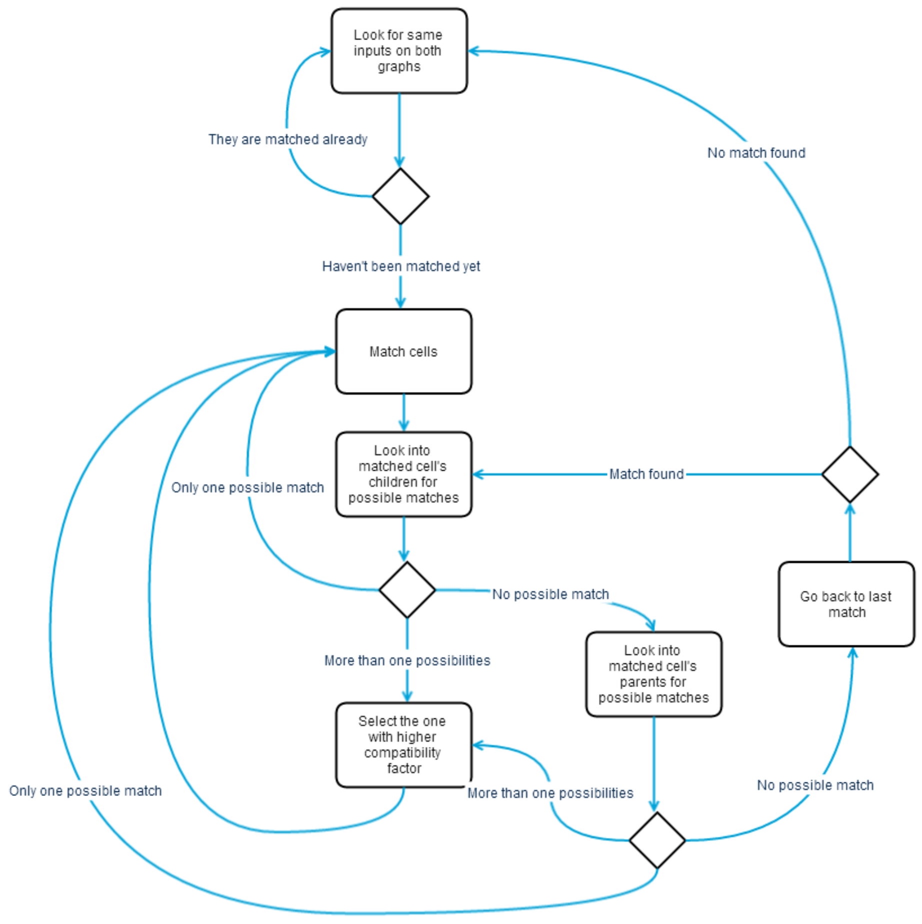

Look for same inputs on both graphs
They are matched already
No match found
Haven't been matched yet
Match cells
Look into matched cell's children for possible matches
Match found
Only one possible match
No possible match
Go back to last match
More than one possibilities
Look into matched cell's parents for possible matches
Select the one with higher compatibility factor
Only one possible match
More than one possibilities
No possible match

Figure 4.1 – UML diagram explaining the algorithm used to match cells.

The first step is to look for same "input" vertices on both graphs. This is done by comparing the "input" list of vertices on each graph which only contain this type of vertices. Then, for each input, a test is made to check if it has already been matched on both graphs.

If it is not the case, they are marked as matched and for each children of matched cells, it starts looking for possible matches for them on every children of the other graph. A list of possible matches is saved, if it has only 1 element, then a match is directly made and moves to the next children. If it has more than 1 element, then, a compatibility factor is calculated and the child with a higher value is chosen.

To better explain it, it is necessary to follow the example on Figure 4.2, it shows how matches are done on one of the graphs and its order. Bear in mind that only matches are represented, because these cells can also have children and parents that weren't matched.

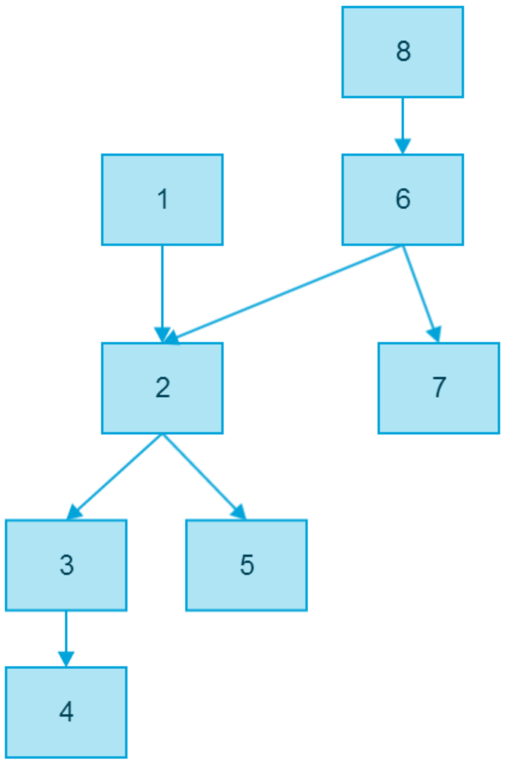


Figure 4.2 - Example of the order that matches are being done.

It starts by looking at the inputs, and if both graphs have the same input, they are matched as it happens on cell 1. Then it looks at their children and compares them, in order to find the best match. The comparison is made by picking a child on the first graph and giving a “Compatibility Factor” to each possible match on the other graph and the match is made with the cell that has the highest factor. This selection procedure is explained with detail further ahead.

Each time a match is made, the search proceeds to its child. For example after looking on cell 2’s children and finding cell 3 as a match, it doesn’t look for more matches of 2’s children. Instead, it goes on, following 3’s path and looking for its children for matches and then matches cell 4. This time no match is possible on 4’s children so it looks for a match on its parents other than the one where it came from and also no match is found so it goes one step back to cell 3. The same happens here, the procedure repeats itself and goes back again, this time to cell 2. Already in 2, there is other child that can be used to match and that’s what happens. Cell 5 is matched and because there is no other match on its path, it goes back to cell 2. Now that all possible matches on cell 2 were made, its parents are checked. Since it came from 1, this isn’t considered and cell 6 is found as a match. A match on 6’s children is found and after another one on its parents is done always following the same method. After cell 8 is matched, there is no other match possible on the graph. This can be easily checked by following the algorithm, the analysis goes all back to cell 1 (through 8->6->2->1) and by reaching it, no other match is found to “go back another step”, the cycle ends. To sum up, by following this method, the route made while analyzing cells was: 1->2->3->4->3->2->5->2->6->7->6->8->6->2->1. At this point, it starts all over again by looking on the inputs for one that hasn’t been matched before and has a match on the other graph. When this point fails, then the matches are completed and the mapping of possible merges between both graphs is complete.

To compare cells and match them, whenever there is more than one possibility, the compatibility factor is used. It was necessary to use this complex approach due to the possibility of having several cells representations of the same operations and therefor also having same address and the only difference would be their inputs and outputs (which have been explained with more detail on section 3.1.1).

The compatibility factor is obtained by comparing all parents and all children of each vertex, and for each correspondence, the factor increases. For constants, inputs and exits it compares their IDs and their Labels (they do not have addresses or instructions). And for operations, addresses and instructions are the attributes compared. After doing this procedure to all possible merges, the one with the best factor is selected. Figure 4.3 shows a merge view example that can help understand this problem and how the solution works and Figure 4.4 contains information about some vertexes of this example.

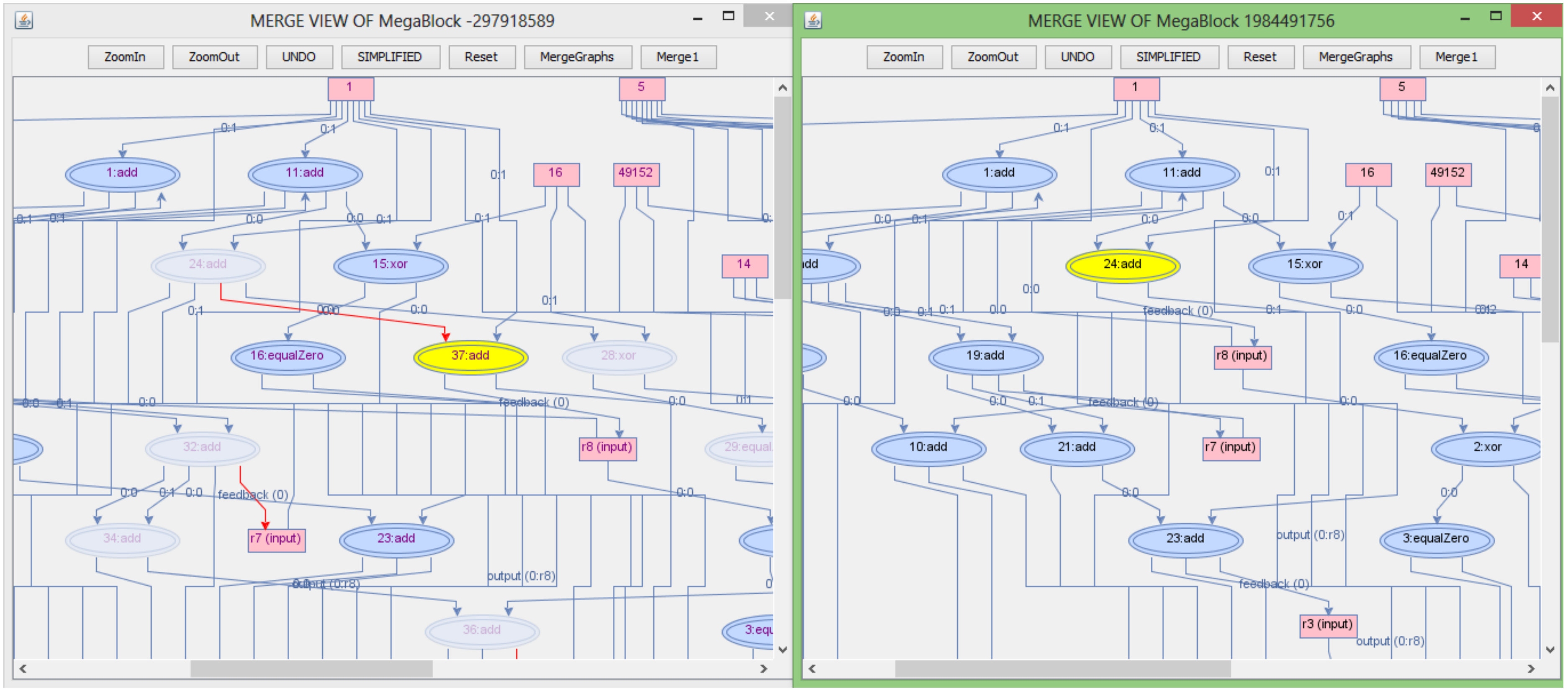
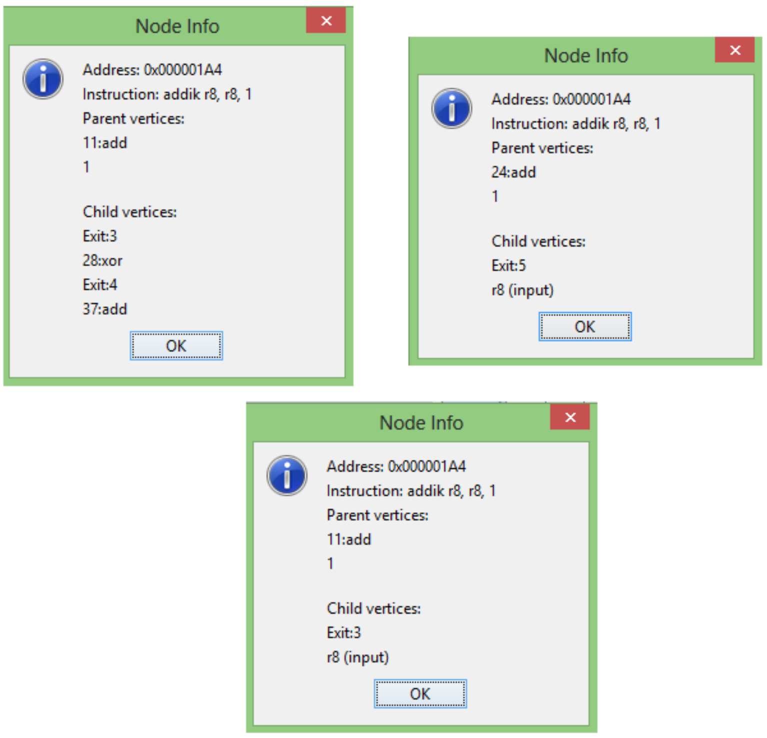


Figure 4.3 - Merge view example.


Node Info

Address: 0x000001A4
Instruction: addik r8, r8, 1
Parent vertices:
11:add
1

Child vertices:
Exit:3
28:xor
Exit:4
37:add

Node Info

Address: 0x000001A4
Instruction: addik r8, r8, 1
Parent vertices:
24:add
1

Child vertices:
Exit:5
r8 (input)

Node Info

Address: 0x000001A4
Instruction: addik r8, r8, 1
Parent vertices:
11:add
1

Child vertices:
Exit:3
r8 (input)

Figure 4.4 – Detailed information of some vertices.

As it can be seen by the yellow colored vertices, “37: add” on the first graph is associated to “24: add” on the second one. The first graph has the exact same operation, “24: add”, and despite that, “37: add” was chosen instead. So, in this case, it started by the constant “1” which was directly linked on both graphs and then, compared all the children of both “1” vertices for the ones that have the exact same instruction. If the first graph would have only one vertex with the instruction “addik r8, r8, 1”, it would have been linked and the procedure would go on through their children but since both “24:add” and “37:add” have the same instruction as “24:add” on second graph, they need to be compared with the compatibility factor. Detailed vertex info on Figure 4.4 show that “24:add” on second graph has 2 parents (“11:add” and “1”) and 2 children (“Exit:3” and “r8(input)”), “37:add” has the same amount and types of cells connected to it while “24:add” on first graph, on its turn, has the same type of parent vertices but it has some differences regarding its children: it has “28:xor”, “Exit:4” and “37:add”, and it is missing an input type of cell. So, vertex “37: add” has a higher factor and it will be chosen over “24: add”.

At this point, all mergeable cells are mapped. To actually merge both graphs, the one with more cells is used to be changed so it can also work as the other graph. Looking back on the last example, graph 1 of Figure 4.3 has a red edge connecting “24: add” to “37: add”. These red edges show where multiplexers are placed to create the merged graph. They are detected by analyzing both graph’s matching and non-matching vertices. Whenever a matching vertex receives data from different sources on each graph, edges that represent that data flow are marked as red. By inserting a multiplexer on that spot, connecting both edges to its inputs and connecting its output to the matching cell, a selection between each input of the multiplexer can be made. Figure 4.5 shows the result of the merge operation that explains it better.

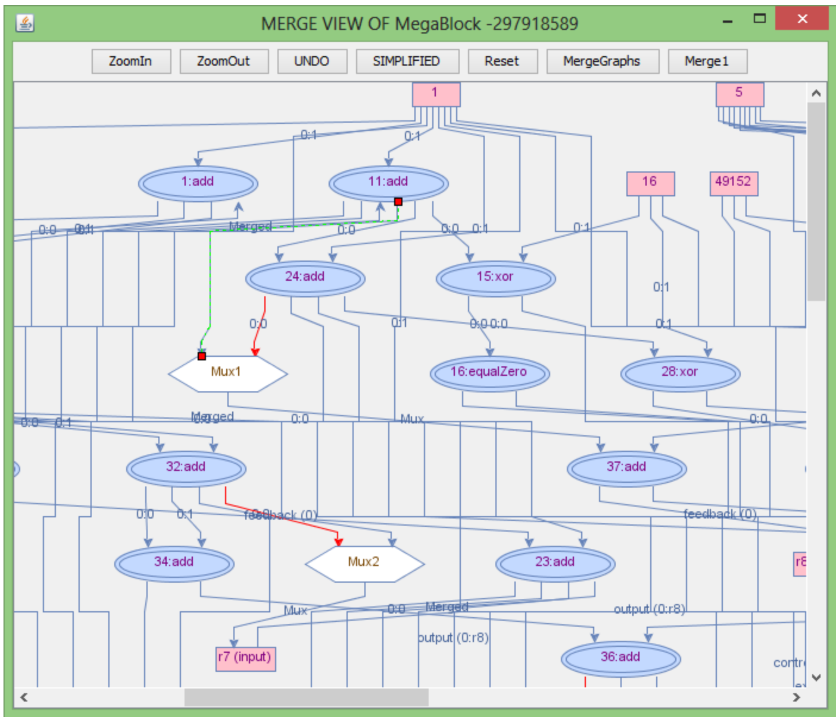


Figure 4.5 - Result of merging both graphs from Figure 4.3.

The difference is that on graph 1, cell “37: add” has cell 1 and cell “24: add” has parents while it’s matching on graph 2, cell “24: add”, has cell 1 and cell “11: add”. This cell “11: add” of graph2 is matched to the same cell “11: add” on graph1 so cell “37: add” will have a multiplexer connected to it to select if either the input comes from “24: add” or “11: add”. In this example, the merged graph needs more multiplexers. As Figure X shows, there is another connected to “r7(input)”. For this Megablock to be able to operate correctly as either one of the graphs that gave origin to ti, it is necessary to make the correct selection on all multiplexers. To achieve this, all edges from graph 1 (the red ones) are connected, as the first input of each multiplexer, while the other edges are its second input. To choose between all inputs 1 or all inputs 2, all multiplexers have the same control bit on its input 3. Figure 4.6 shows where this control bit comes from.

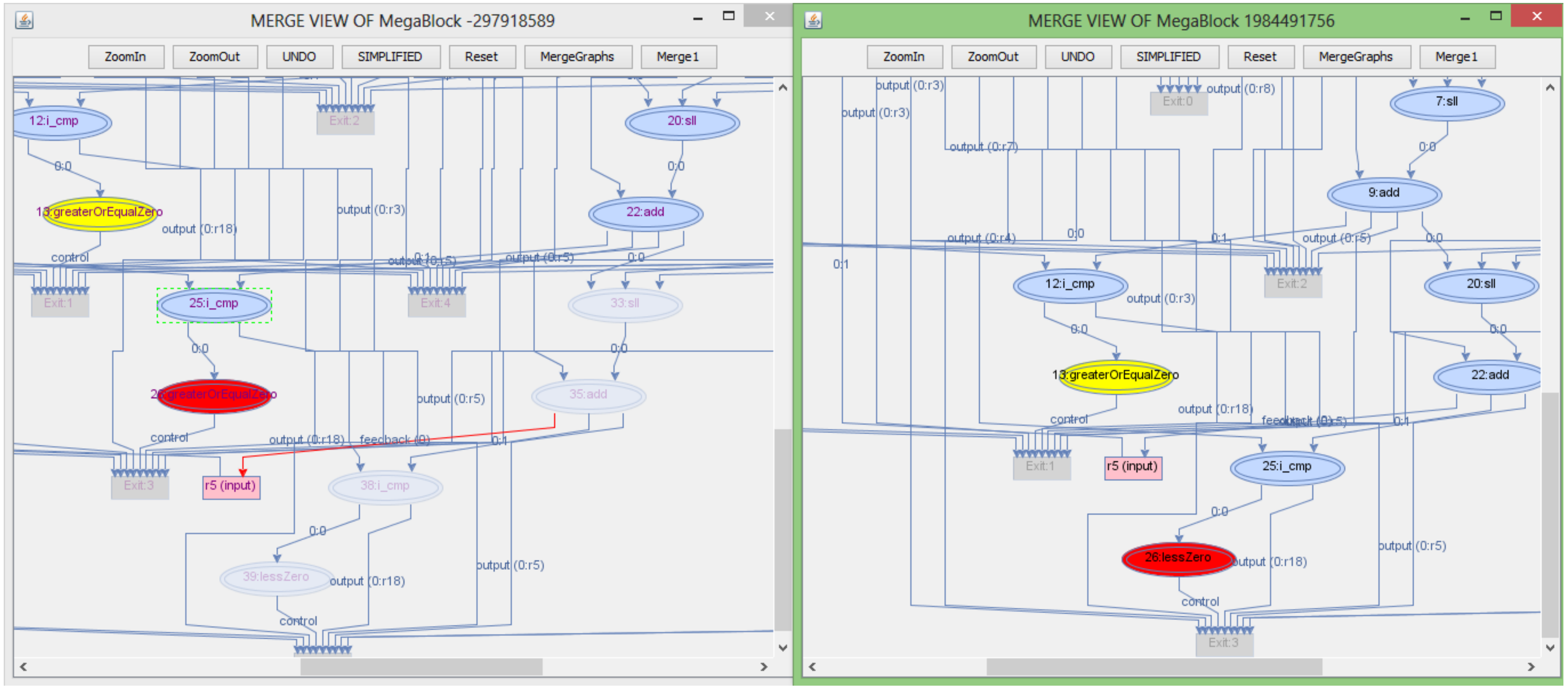


Figure 4.6 - Portion of a merge view showing the control vertex as red.

Those red cells represent a special case. They are both the same operation and they have both the same parent and the same child but, in fact they have a different label. In graph 1 it's "*greaterOrEqualZero*" and on graph 2 it's "*lessZero*". This means that they represent opposite conditions and the control bit they generate (that goes to the exit) is never the same on both graphs. This is where the control bit comes from, the merged graph only shows 2 inputs on each multiplexer, but this bit is actually connected as its 3rd input. It just isn't shown because it would increase the complexity of the graph and that becomes less pleasant to visualize. In this way it is possible to control all multiplexers to have the same input at the same time, making the merged graph to execute as either graph1 or graph2.

Chapter 5

Validation and Results

5.1 - GUI

By comparing several graph representations of megablocks created by this software with their respective “asm” files, it is possible to conclude that there is always a correct representation of all the operations contained on each Megablock and all the edges between them. In fact, its interactivity also works properly and vertices and edges can be moved around causing no issue. The popup menu is only accessible on the main graph by right clicking a Megablock as intended and, as checked, the options on the menu show no problem. The top panel shows the correct buttons according to the type of graph shown (main graph, Megablock’s content, merge view, simplified) and they all work perfectly. The only problem detected is related with the fact that it hasn’t been possible to undo a merge due to its complexity. The display and organization of Megablock contents could be better indeed. With a big amount of vertices and edges, their labels and also the cells where the edges are connected can be hard to see. However, this is a limitation that comes from the use of jgraph as it only contains few layout types available and this is the result of the hierarchical layout. Additional features like Zoom in, Zoom out and double click to show the cell’s parents/children were implemented to help to get a better understanding of how each Megablock works.

Following is a more detailed description of the tool’s usage. The tool starts up by showing a representation of the Megablocks of the last program executed in the Megablock Extractor as shown on Figure 5.1.

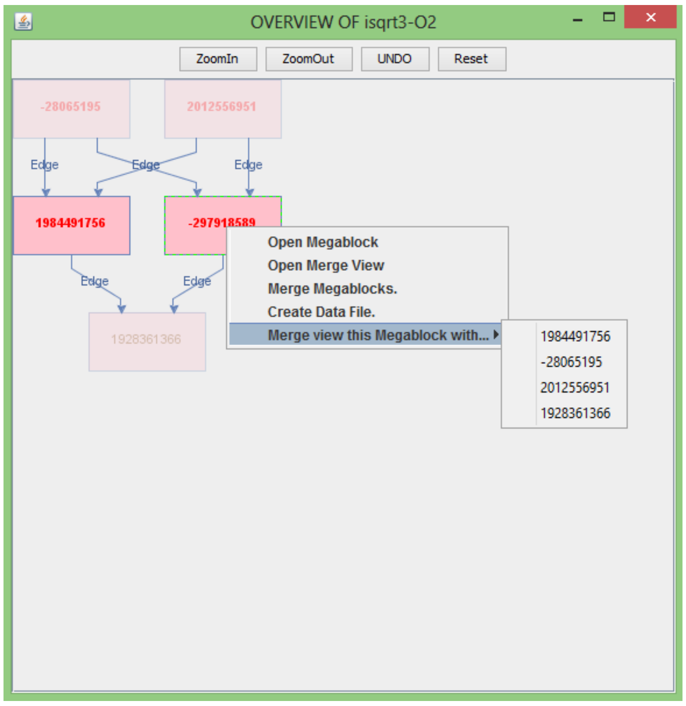


Figure 5.1 - Main graph with popup menu opened

They are ordered from the top to the bottom by the starting position in the memory of each one, in such way that the MegaBlocks with lower address are positioned on the top and the ones with higher are on the bottom. The edges might mislead to a false interpretation about a possible relation between MegaBlocks or even an execution order but this is not the case. They connect MegaBlocks to the next one(s) in this order of addresses. With this view, MegaBlocks with the same starting address are next to each other horizontally and might be good candidates for a merge.

By hovering the mouse cursor over a Megablock, it is highlighted along with all the others that have the same starting address. And by double clicking it, another window opens showing its content.

The panel on the very top has a couple of buttons which allows to “Zoom in”, “Zoom out”, “Undo” changes done on the graph and “Reset” the view to the starting one. The zoom buttons are useful on cases when, for instance, there are so many vertices that they can’t fit on the original window. If a user drags to many cells around, he can use the “Undo” button to reverse the last action or the “Reset” to return the view to the starting point.

There is also a popup menu that can be accessed by right clicking on a Megablock. This menu allows to “Open MegaBlock” which performs the same action as double clicking it, “Open Merge view” opens both the selected Megablock and one with the same address side-by-side allowing to compare both and to perform a merge if wanted. “Merge Megablocks” immediately merges the selected Megablock with one with the same address without opening the merge view, “Create Data File” creates a file with the parsablegraph format like the one created by the Megablock Extractor, but in case of applying to a merged block, it has all the changes included addition of multiplexers, “Merge view this Megablock with…” performs the “Open merge view” action with a chosen Megablock.

The inside view of a Megablock accessed either by a double-click or the "Open Megablock" option, on the right-click menu, brings up a window representing the content of the Megablock. An example of the inside view of a Megablock is visible on Figure 5.2.

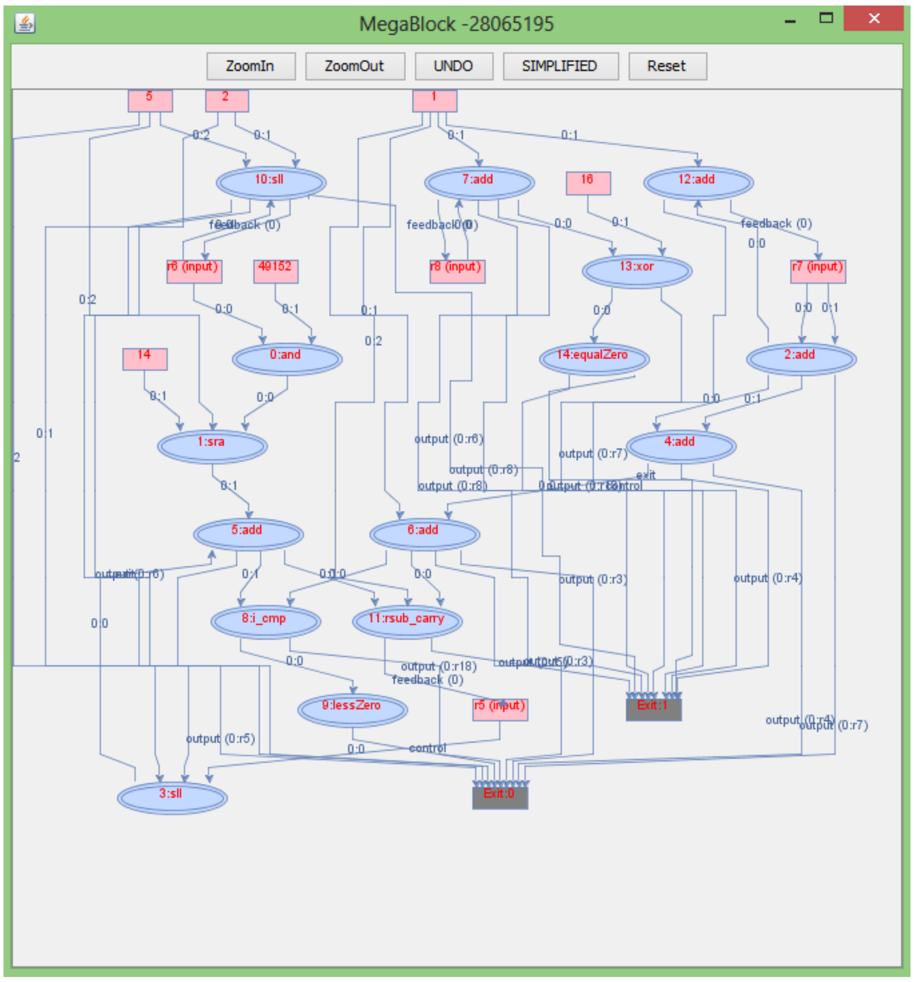
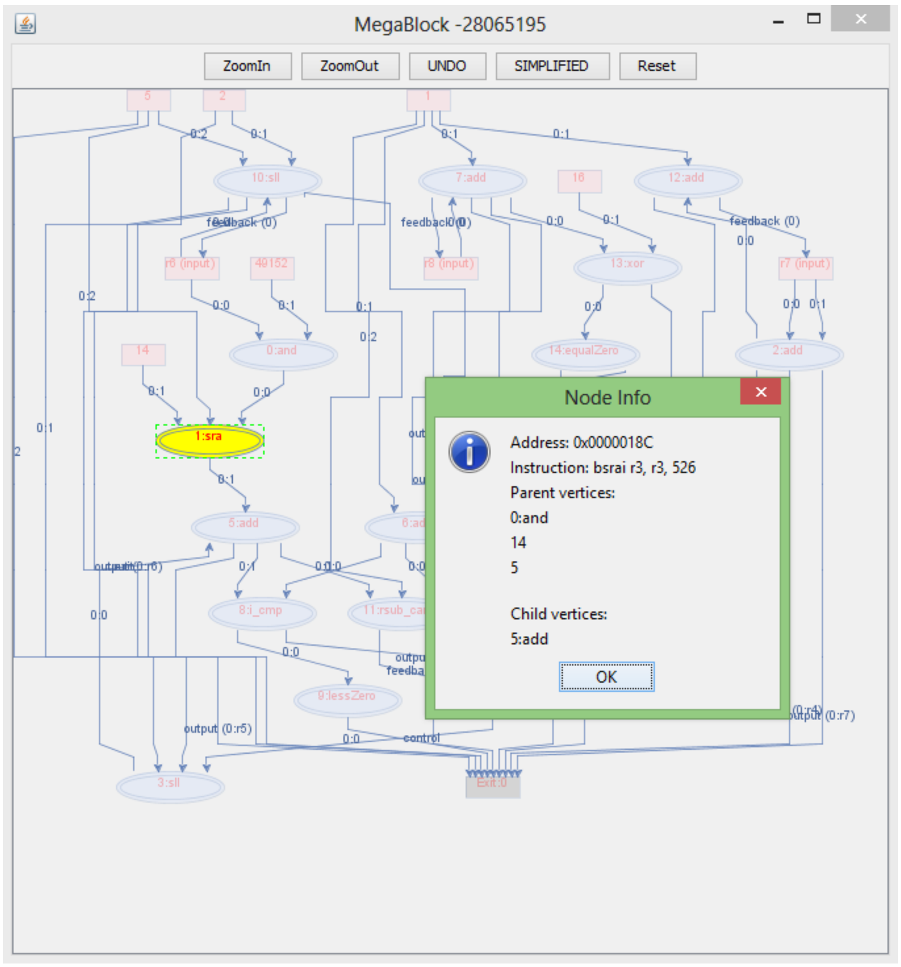


Figure 5.2 - Inside view of a Megablock

Inputs are represented by pink rectangles, constants have their value shown on the label and only have outgoing edges, registers can have both incoming and outgoing edges as they can be read or/and written from.

Exit points are shown as grey rectangles and because they lead to the end of the Megablock processing, only incoming edges are connected to them.

The blue elliptical cells represent operations. Each one has a different number which was assigned by the Megablock Extractor and doesn't have any meaning, it only serves to better identify and reference each particular operation vertex. Followed by the number is the operation type of the vertex. Despite of having, all of them, different identification numbers, a very same operation can be, and it is most of the times due to MegaBlock's repetitive nature, represented more than once. These vertices have the same operation and process on the same inputs but since they are (or at least one of them is) obtained from a different vertex, it is represented again on the graph. Detailed information about an operation can be obtained by simply double clicking on it. This pops up a window with its address, complete assembly instruction and parent/child vertices that are connected to it if any. Figure 5.3 contains an example.



Address: 0x0000018C
Instruction: bsrai r3, r3, 526
Parent vertices:
0:and
14
5

Child vertices:
5:add

Figure 5.3 - Detailed information about a vertex.

In case of viewing a Megablock that results from a merge, white hexagons are shown as representing multiplexers that allow the Megablock to operate as any of the two ones that led to this merge. Another difference is the presence of an operation colored red instead of blue. This operation is the one that controls the selections of all multiplexers. It generates a control bit that is connected to each multiplexer but it isn't shown on the graph to not overwhelm it with more edges.

On this view, the top panel has an additional button: "SIMPLIFIED". This shows another view of the current graph without inputs and exits. Even though, it loses most of its information, because some graphs can have too many vertices and, by restricting the view to operations only, it can help to understand how they are connected and also their dependencies. This can be seen on Figure 5.4.

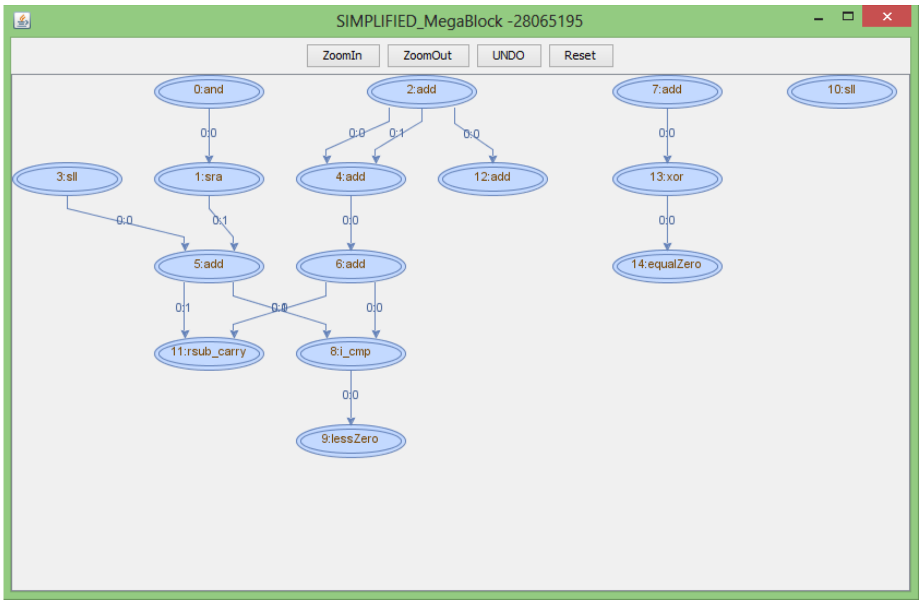


Figure 5.4 - Simplified view.

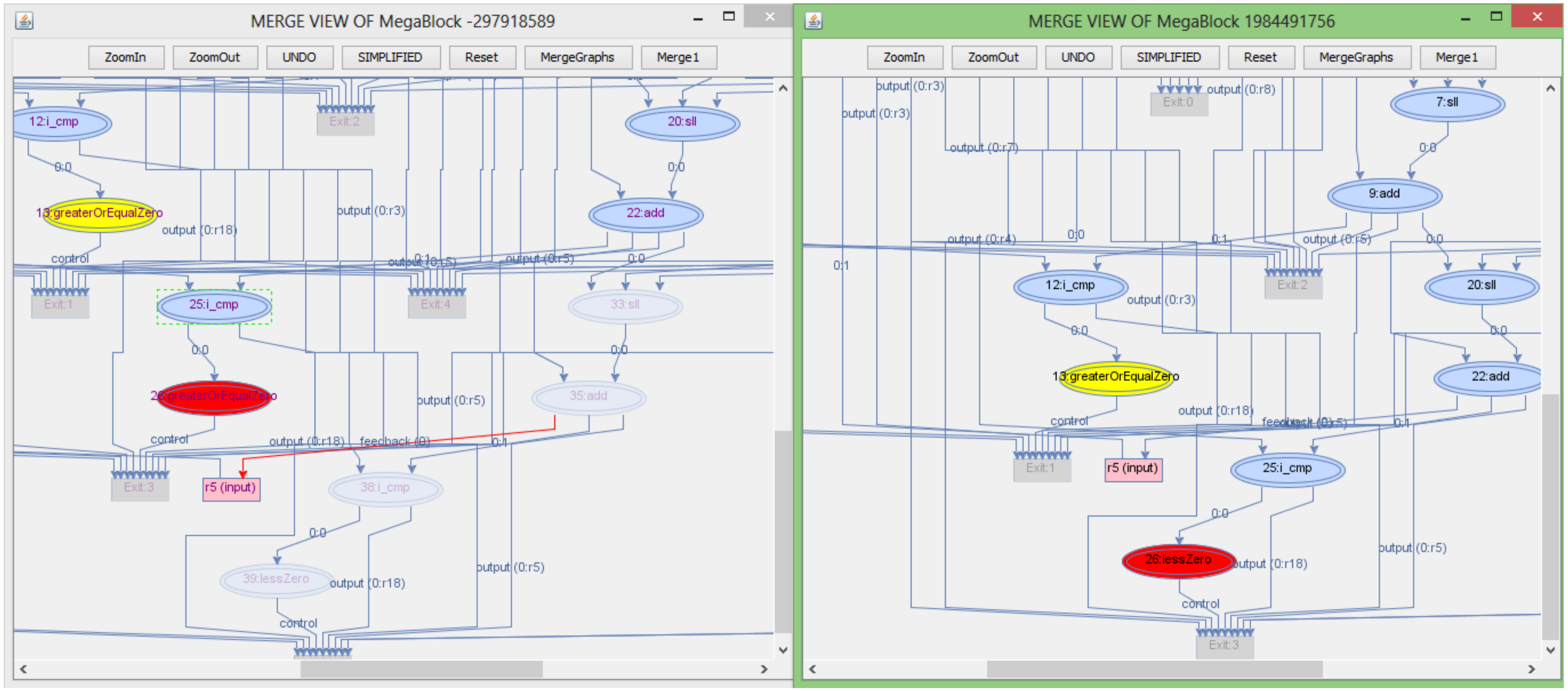


Figure 5.5 - Merge view.

The “Merge View” shows the two MegaBlocks contents side-by-side as displayed on Figure 5.5. The graph on the left is the one that will register changes done in case of a merge. This is the Megablock that has more operation vertices which usually makes it the best one to create the merge from. In this view, both graphs have a red colored operation indicating the place where the control bit for the multiplexers is going to be generated. The red edges indicate edges that will need to have a multiplexer and another edge connected to it according to the other graph.

In this view, hovering over a vertex changes its color to yellow as well as its corresponding vertex on the other graph. It also highlights the group of mergeable vertices that they both belong to by giving a good perception of common vertices to both graphs that can be merged. The “Merge” button on the top panel also highlights these vertices. If there are more group associations, more buttons are added to allow the highlight of different correspondences between both graphs. The panel on top also has a `MergeGraphs` button to perform that exact action by making the needed adjustments on the graph on the left side and to transform it into the merged graph and to be able to process both graphs. The representations of these megablocks on the main graph are also merged and a new Megablock named “*megablock1* MERGED WITH *megablock2*” (where *megablock1* and *megablock1* are the Megablock IDs of each one) replaces both of them. Due to its complexity, the UNDO button has no effect on this action and the merged megablock cannot be merged with another Megablock but it is still possible to see its content and detailed information of each vertex.

5.2 - Validation of the Merged Block

Initially, the procedure to validate the merge ability was to use the “parsablegraph.txt” created from a merged Megablock, on the embedded system that Nuno Paulino uses in his work. But the results from MegaBlock Extractor don’t include any multiplexer and since this is a new type of vertex and it has particularities such as a control bit and a selection between two inputs that needs to be done. At the time this work was done, the system wasn’t prepared to receive and to implement this on its hardware. So, it was decided that validation would be done by using ModelSim to simulate hardware that would correspond to the merged megablock.

The steps were starting by creating simple programs in “c”, compiling them with gcc to create the “.elf” file, then loading them into “MegablockExtractor” to finally generate all output files, running MegablockMultipath and merging two megablocks and finally using ModelSim to make a model of the resulting merge. At the compiling step, the program’s variables final values are obtained and then compared with the simulation. By comparing these values, conclusions can be made such as if the merge block can correctly work as both megablocks or not.

On the attempt to use this method to validate the merge structure, very simple codes were made and Figure 5.1 shows an example.

```
void main() {
       int A;
       for (A = 0; A < 10; A++){
              if(A<5)     {
                     A ++;
              }
       }
}
```

Figure 5.6 - Source code of an example to test.

An “if” inside a “for” makes this case to have a loop and several decisions. Each time the loop is done, “A” is incremented and in case of being lower than 5, it is incremented again. This example resulted into two MegaBlocks with the same address and with the possibility to be merged. Figure 5.7 shows the assembly code for one of those MegaBlocks while Figure 5.8 shows its graph representation obtained with the `MegablockMultipath`.

```
0x00000200 lwi r4, r19, 4  -> 0:add
                              1:load
!0x00000204 addik r3, r0, 4 -> Removed
0x00000208 cmp r18, r4, r3 -> 3:i_cmp
0x0000020C blti r18, 16    -> 4:lessZero
0x00000210 lwi r3, r19, 4  -> 5:add
                              6:load
0x00000214 addik r3, r3, 1 -> 7:add
0x00000218 swi r3, r19, 4  -> 8:add
                              9:store
0x0000021C lwi r3, r19, 4  -> 10:add
                              11:load
0x00000220 addik r3, r3, 1 -> 12:add
0x00000224 swi r3, r19, 4  -> 13:add
                              14:store
0x00000228 lwi r4, r19, 4  -> 15:add
                              16:load
!0x0000022C addik r3, r0, 9 -> Removed
0x00000230 cmp r18, r4, r3 -> 18:i_cmp
0x00000234 bgei r18, -52   -> 19:lessZero
```

Figure 5.7 - Assembly code resulting from using the source code of Figure 5.1 on the Megablock Extractor.

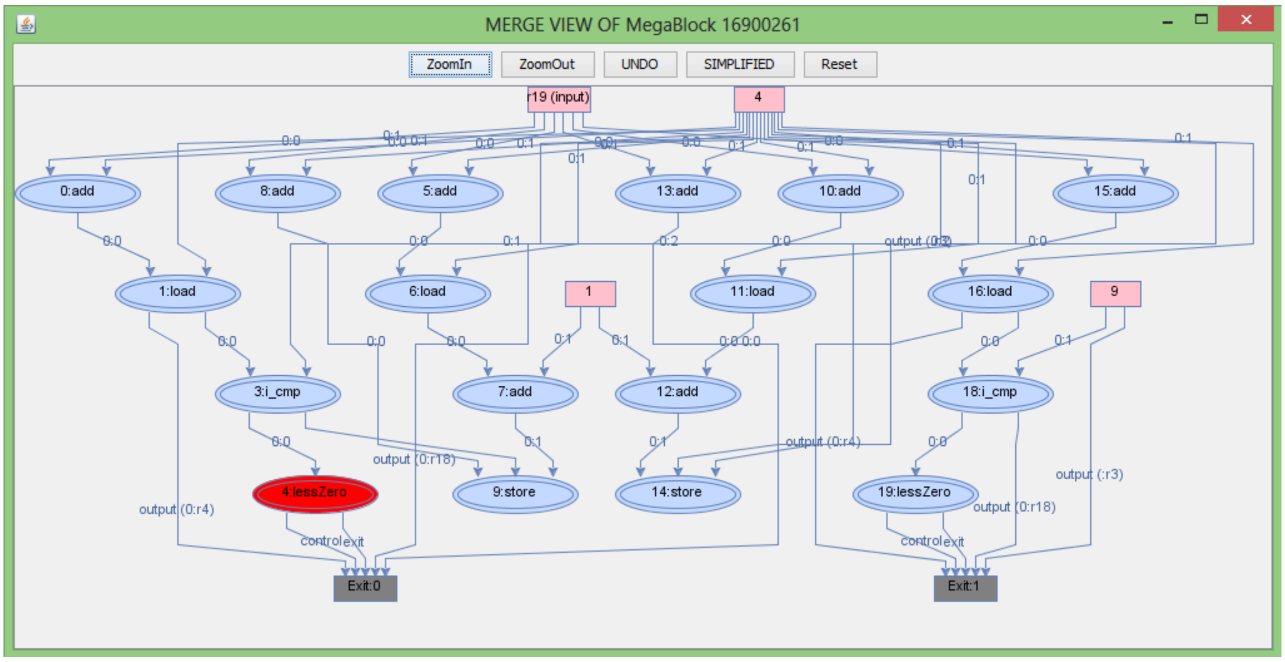


Figure 5.8 - Graph representing operations from Figure 5.1

A simple example like this one is represented by a quit complicated graph with more than 20 vertices and 40 edges. This led into struggles to create a model and unfortunately, validation was not possible.

A flaw was also found with this case and it can be seen on Figure 5.9.

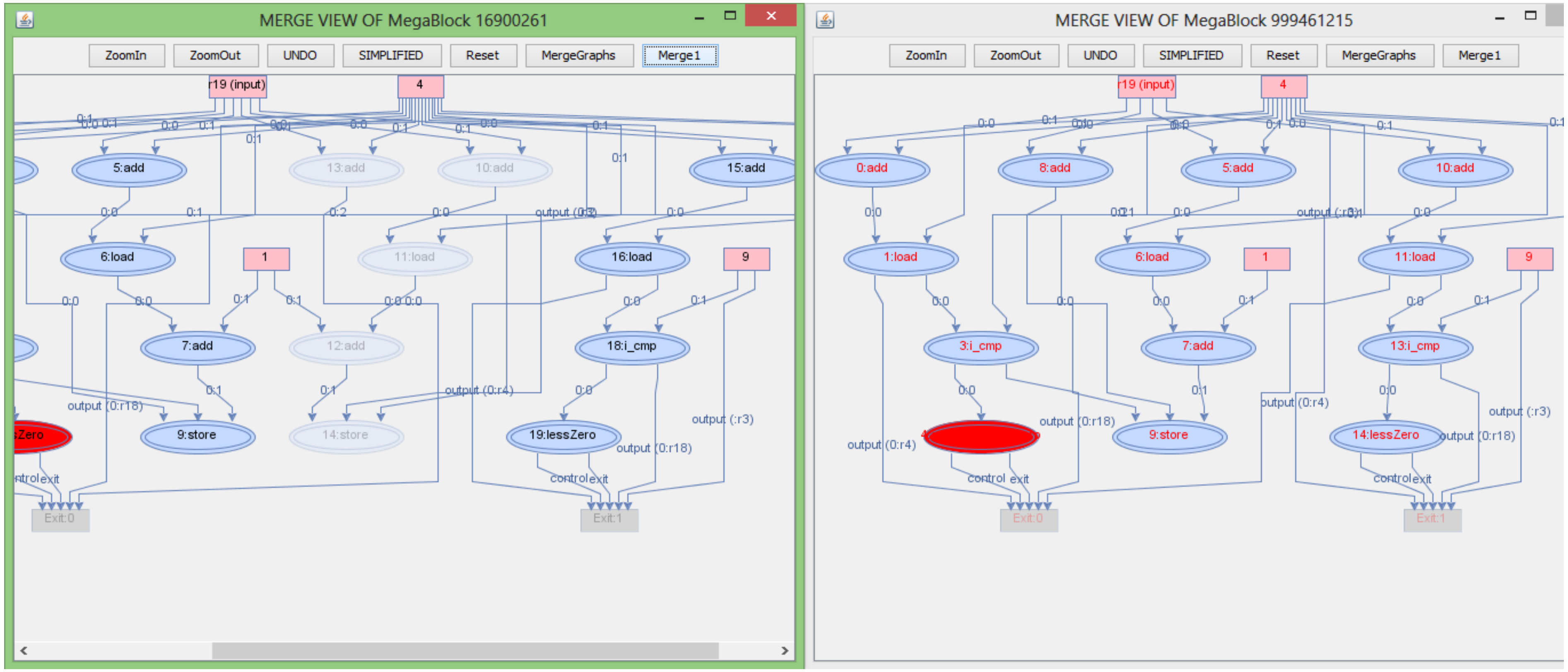


Figure 5.9 - Example of a flaw on the merge technique.

All vertices on the graph on the right side are matched on the left one but this left one has some more vertices that aren't on the graph of the right side, these are the ones that aren't highlighted. The merge algorithm adds multiplexers between a non-mergeable vertex and its mergeable child as explained on chapter 4, this selects the mergeable cell input. In this case, there isn't any non-mergeable vertex with a mergeable child because these operations end on the "14: store" operation that doesn't even have a child. To correct this, that "14: store" has to be limited on the merged Megablock and to only be able to process when the graph on the left is selected, which can still be checked by the control bit from the red colored vertex.

Chapter 6

Conclusions and future work

This dissertation has its basis on other previous work done on embedded systems consisting of a GPP and a CGRA with focus to boost its overall performance. As embedded systems are very common on most of the technological equipment that we need to deal on a daily basis, evolution on this matter have a big impact on a very wide range of types of electronic products which result into evolution on several different areas. CGRAs are able to process data faster than a GPP so an effective method to map instructions into it is desirable. This means finding high execution program traces, which is already done by the detection of Megablocks. The aim of this dissertation is to aid by having a good overview of the programs to execute in an embedded system with usage of the MegaBlock concept and to try to boost its performance by exploring a weak spot on the mapping technique used and for which a Merge of Megablocks would have high impact on improving its performance.

The work done on this dissertation shows that a tool was developed and it correctly shows MegaBlock's contents and allows to interact with them. It also allows doing the Merge of Megablocks. Though it couldn't be validated as explained on chapter 5, the algorithm used (chapter 4) can be of good use since it focuses on a careful analysis of the relationships between vertices and at the same time comparing some of their characteristics. So the objective to make an interactive tool was fulfilled, while the objective of developing the merge Megablock technique was also done but still unable to be validated and a flaw was detected. Generation of the VHDL code wasn't implemented but on the other hand, the tool generates an output file in parsablegraph format that allows the receiver of this work to use it on its hardware. As for additional goals, the tool does not allow having a nested view on the Megablocks but it does allow to have an overview of them and to pick which one to open and show its content. The tool does not allow recovering a work previously finished but has the ability to undo the interactions done with the exception of Megablock's merging due to its complexity.

Future work should include improvements on this tool to overcome the flaw on the merge ability and to develop it further in order to be able to merge more than 2 Megablocks and even to do some other operations that can improve performance. It would also be very interesting to expand the merge technique to another kind of approach that would allow the execution of the same Megablock on more than 1 CGRA at a time or even spreading some execution between them so that more than one could be processing at the same time.

References

[1] N. Paulino, "Transparent Generation of Reconfigurable Hardware at Runtime from Execution Traces," 2013.

[2] J. Bispo and J. M. P. Cardoso, "On Identifying and Optimizing Instruction Sequences for Dynamic Compilation," *Proceedings of the International Conference on Field-Programmable Technology,* pp. 1-440, 2010.

[3] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, “Transparent reconfigurable acceleration for heterogeneous embedded applications,” in Proceedings of the Conference on Design, Automation and Test in Europe (DATE ’08), pp. 1208–1213, Munich, Germany, March 2008.

[4] A. C. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, “Run-time adaptable architectures for heterogeneous behavior embedded systems,” in Proceedings of the 4th International Workshop Reconfigurable Computing: Architectures, Tools and Applications, pp. 111–124, 2008.

[5] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner, “An architecture framework for transparent instruction set customization in embedded processors,” in Proceedings of the 32nd Interntional Symposium on Computer Architecture (ISCA ’05), pp. 272–283, June 2005.

[6] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, “Application-specific processing on a general-purpose core via transparent instruction set customization,” in Proceedings of the 37th International Symposium on Microarchitecture (MICRO’04), pp. 30–40, Portland, Ore, USA, December 2004.

[7] J. Bispo, N. Paulino, J. M. P. Cardoso, and J. C. Ferreira, "Transparent Runtime Migration of Loop-Based Traces of Processor Instructions to Reconfigurable Processing Units," *International Journal of Reconfigurable Computing,* pp. 1-20, 2013.

[8] R. Lysecky and F. Vahid, “Design and implementation of a MicroBlaze-based warp processor,” Transactions on Embedded Computing Systems, vol. 8, no. 3, article 22, 2009.

[9] R. Lysecky, G. Stitt, and F. Vahid, “Warp processors,” ACM Transactions on Design Automation of Electronic Systems, vol. 11, no. 3, pp. 659–681, 2006.

[10] A. Mehdizadeh, B. Ghavami, M. S. Zamani, H. Pedram, and F. Mehdipour, “An eficient heterogeneous reconfigurable functional unit for an adaptive dynamic extensible processor,” in Proceedings of the IFIP International Conference on Very Large Scale Integration (VLSI-SoC ’07), pp. 151–156, October 2007.

[11] H. Noori, F. ehdipour, K. Murakami, K. Inoue, and M. S. Zamani, “An architecture framework for an adaptive extensible processor,” Journal of Supercomputing, vol. 45, no. 3, pp. 313–340, 2008.

[12] J. Bispo, "Mapping Runtime-Detected Loops from Microprocessors to Reconfigurable Processing Units," 2012.

[13] J. Bispo and J. M. P. Cardoso, "On Identifying Segments of Traces for Dynamic Compilation," *International Conference on Field Programmable Logic and Applications,* pp. 263-266, 2010.

[14] "<SUPERBLOCK.pdf>."

[15] *Matplotlib*. Available: http://matplotlib.org/

[16] *Graphviz*. Available: http://www.graphviz.org/

[17] *Pydot*. Available: https://code.google.com/p/pydot/

[18] *Xdot*. Available: https://github.com/jrfonseca/xdot.py

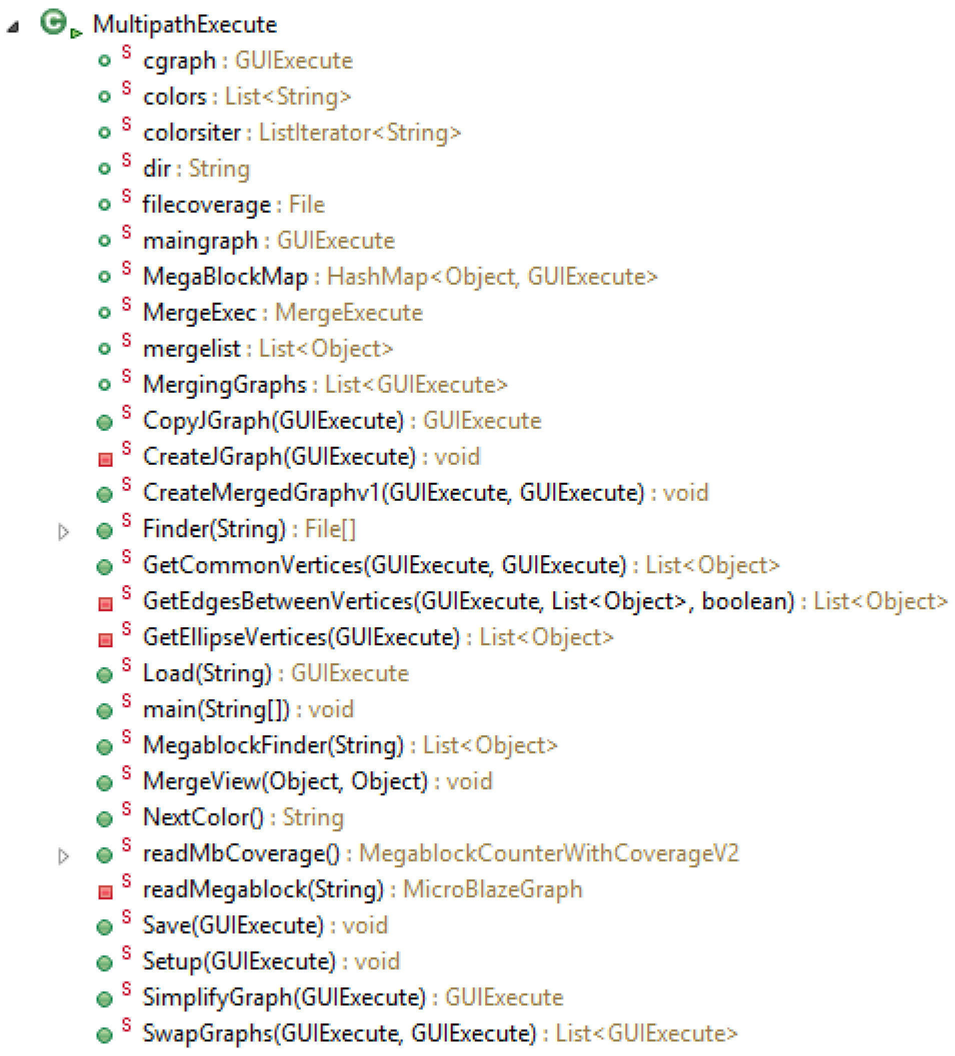
[19] *Python-Graph*. Available: https://code.google.com/p/python-graph/
[20] *Networkx*. Available: http://networkx.github.io/
[21] *NodeBox*. Available: http://nodebox.net/
[22] *Igraph*. Available: http://igraph.sourceforge.net/
[23] *Python Repository*.Available:http://www.lfd.uci.edu/~gohlke/pythonlibs/
[24] *Jython*. Available:http://www.jython.org/
[25] M. LIU, "Nest-loop transformation techniques considering timing and memory optimization for embedded systems," Doctor of Philosophy in Computer Science, University of Texas, 2006.
Graph Transversals. Available: http://basudip.hubpages.com/hub/Graph_Traversals

Appendixes

These figures show all components that are part of the application developed in this dissertation. Each class has some elements that can be considered the main ones while others aren't so important and serve to help the main ones. Therefore, the architecture of the Multipath Merging Tool was simplified on chapter 3. Here on this section can be found figures illustrating the complete architecture of each class.

Appendix 1

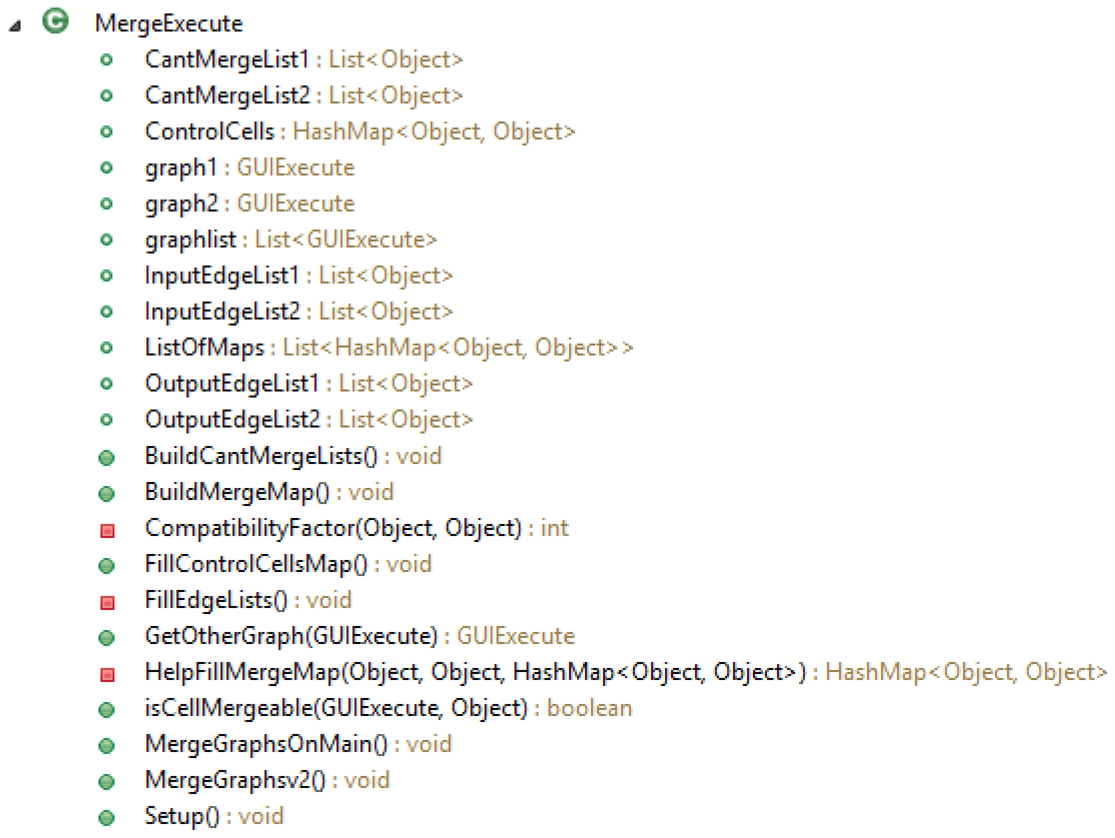
Complete architecture of the `MultipathExecute` class:



```
MultipathExecute
    S cgraph : GUIExecute
    S colors : List<String>
    S colorsiter : ListIterator<String>
    S dir : String
    S filecoverage : File
    S maingraph : GUIExecute
    S MegaBlockMap : HashMap<Object, GUIExecute>
    S MergeExec : MergeExecute
    S mergelist : List<Object>
    S MergingGraphs : List<GUIExecute>
    S CopyJGraph(GUIExecute) : GUIExecute
    S CreateJGraph(GUIExecute) : void
    S CreateMergedGraphv1(GUIExecute, GUIExecute) : void
    S Finder(String) : File[]
    S GetCommonVertices(GUIExecute, GUIExecute) : List<Object>
    S GetEdgesBetweenVertices(GUIExecute, List<Object>, boolean) : List<Object>
    S GetEllipseVertices(GUIExecute) : List<Object>
    S Load(String) : GUIExecute
    S main(String[]) : void
    S MegablockFinder(String) : List<Object>
    S MergeView(Object, Object) : void
    S NextColor() : String
    S readMbCoverage() : MegablockCounterWithCoverageV2
    S readMegablock(String) : MicroBlazeGraph
    S Save(GUIExecute) : void
    S Setup(GUIExecute) : void
    S SimplifyGraph(GUIExecute) : GUIExecute
    S SwapGraphs(GUIExecute, GUIExecute) : List<GUIExecute>
```

Appendix 2

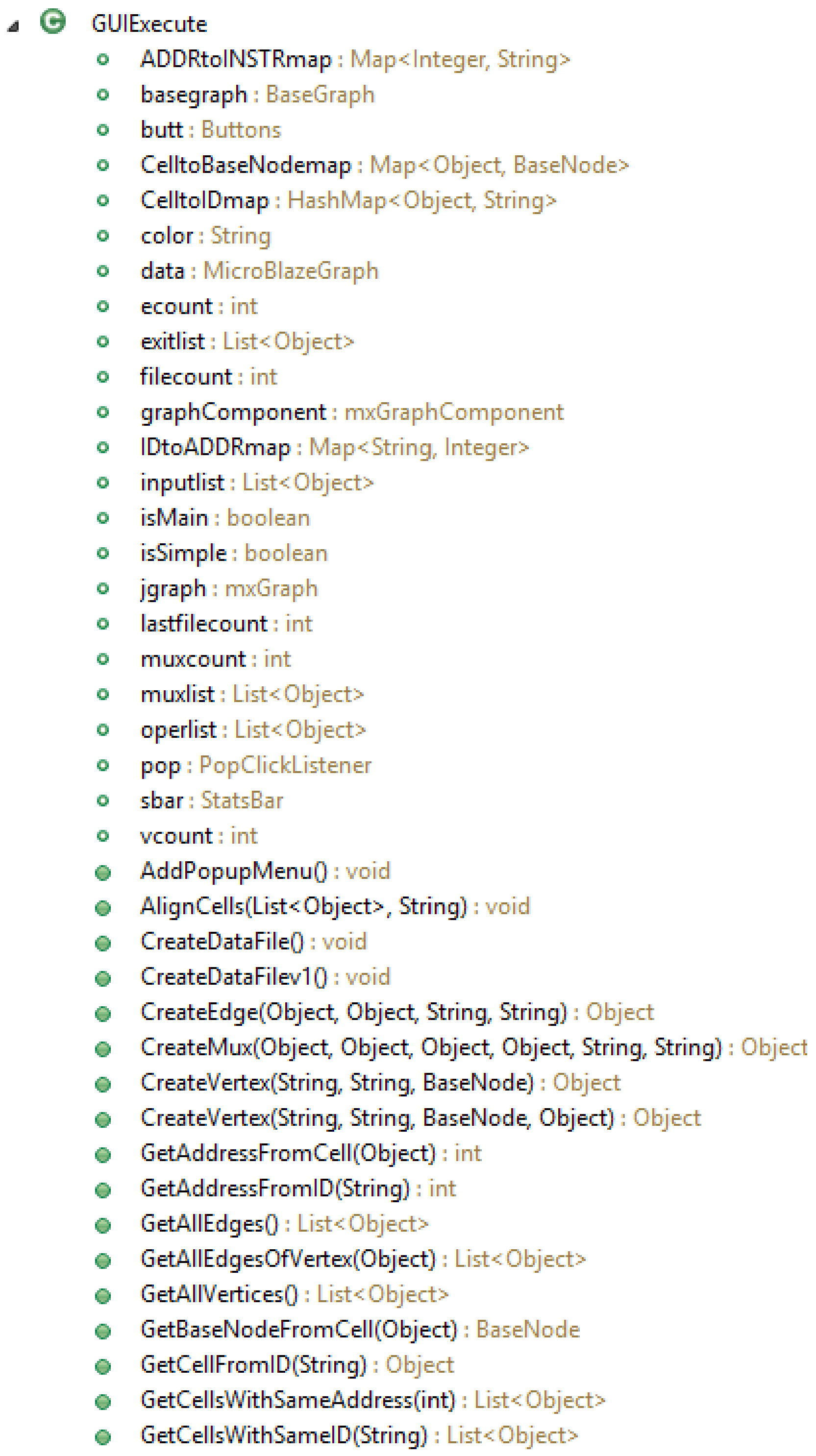
Complete architecture of the `MergeExecute` class:



```
MergeExecute
    CantMergeList1 : List<Object>
    CantMergeList2 : List<Object>
    ControlCells : HashMap<Object, Object>
    graph1 : GUIExecute
    graph2 : GUIExecute
    graphlist : List<GUIExecute>
    InputEdgeList1 : List<Object>
    InputEdgeList2 : List<Object>
    ListOfMaps : List<HashMap<Object, Object>>
    OutputEdgeList1 : List<Object>
    OutputEdgeList2 : List<Object>
    BuildCantMergeLists() : void
    BuildMergeMap() : void
    CompatibilityFactor(Object, Object) : int
    FillControlCellsMap() : void
    FillEdgeLists() : void
    GetOtherGraph(GUIExecute) : GUIExecute
    HelpFillMergeMap(Object, Object, HashMap<Object, Object>) : HashMap<Object, Object>
    isCellMergeable(GUIExecute, Object) : boolean
    MergeGraphsOnMain() : void
    MergeGraphsv2() : void
    Setup() : void
```

Appendix 3a

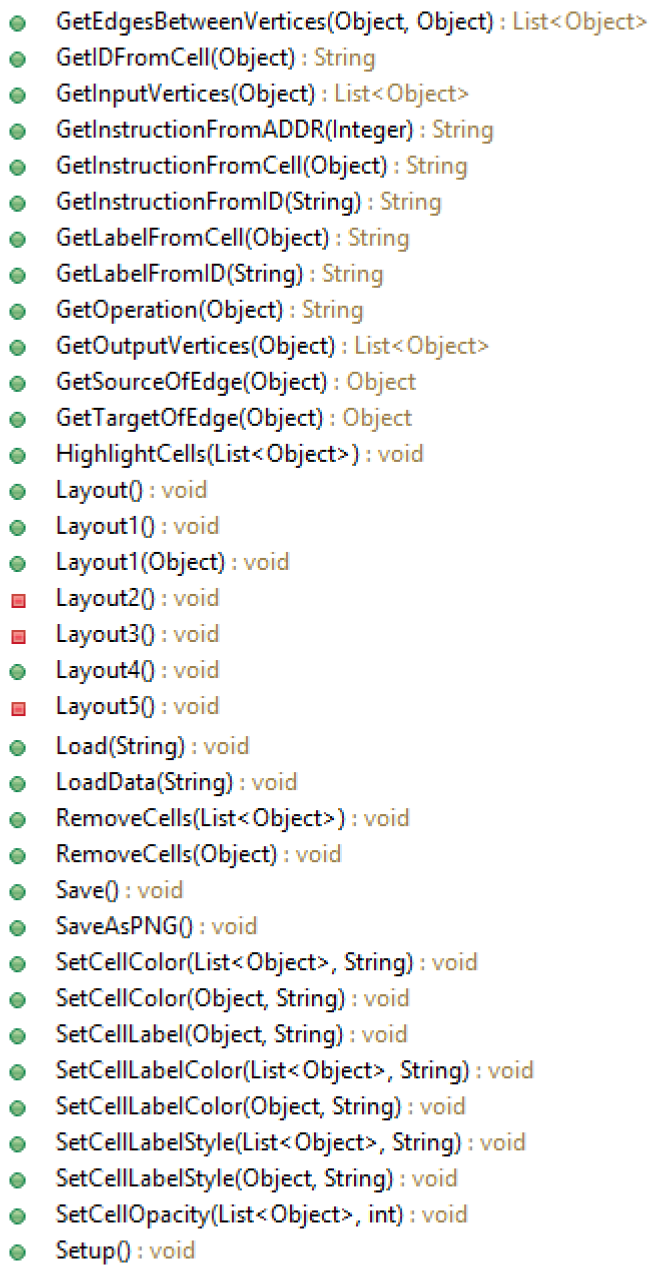
Complete architecture of the `GUIExecute` class (part 1):



```
GUIExecute
    ADDRtoINSTRmap : Map<Integer, String>
    basegraph : BaseGraph
    butt : Buttons
    CelltoBaseNodemap : Map<Object, BaseNode>
    CelltoIDmap : HashMap<Object, String>
    color : String
    data : MicroBlazeGraph
    ecount : int
    exitlist : List<Object>
    filecount : int
    graphComponent : mxGraphComponent
    IDtoADDRmap : Map<String, Integer>
    inputlist : List<Object>
    isMain : boolean
    isSimple : boolean
    jgraph : mxGraph
    lastfilecount : int
    muxcount : int
    muxlist : List<Object>
    operlist : List<Object>
    pop : PopClickListener
    sbar : StatsBar
    vcount : int
    AddPopupMenu() : void
    AlignCells(List<Object>, String) : void
    CreateDataFile() : void
    CreateDataFilev1() : void
    CreateEdge(Object, Object, String, String) : Object
    CreateMux(Object, Object, Object, Object, String, String) : Object
    CreateVertex(String, String, BaseNode) : Object
    CreateVertex(String, String, BaseNode, Object) : Object
    GetAddressFromCell(Object) : int
    GetAddressFromID(String) : int
    GetAllEdges() : List<Object>
    GetAllEdgesOfVertex(Object) : List<Object>
    GetAllVertices() : List<Object>
    GetBaseNodeFromCell(Object) : BaseNode
    GetCellFromID(String) : Object
    GetCellsWithSameAddress(int) : List<Object>
    GetCellsWithSameID(String) : List<Object>
```

Appendix 3b

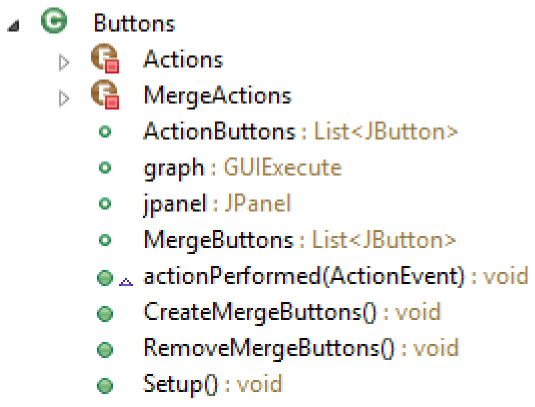
Complete architecture of the `GUIExecute` class (part 2):



- ● GetEdgesBetweenVertices(Object, Object) : List<Object>
- ● GetIDFromCell(Object) : String
- ● GetInputVertices(Object) : List<Object>
- ● GetInstructionFromADDR(Integer) : String
- ● GetInstructionFromCell(Object) : String
- ● GetInstructionFromID(String) : String
- ● GetLabelFromCell(Object) : String
- ● GetLabelFromID(String) : String
- ● GetOperation(Object) : String
- ● GetOutputVertices(Object) : List<Object>
- ● GetSourceOfEdge(Object) : Object
- ● GetTargetOfEdge(Object) : Object
- ● HighlightCells(List<Object>) : void
- ● Layout() : void
- ● Layout1() : void
- ● Layout1(Object) : void
- 🟥 Layout2() : void
- 🟥 Layout3() : void
- ● Layout4() : void
- 🟥 Layout5() : void
- ● Load(String) : void
- ● LoadData(String) : void
- ● RemoveCells(List<Object>) : void
- ● RemoveCells(Object) : void
- ● Save() : void
- ● SaveAsPNG() : void
- ● SetCellColor(List<Object>, String) : void
- ● SetCellColor(Object, String) : void
- ● SetCellLabel(Object, String) : void
- ● SetCellLabelColor(List<Object>, String) : void
- ● SetCellLabelColor(Object, String) : void
- ● SetCellLabelStyle(List<Object>, String) : void
- ● SetCellLabelStyle(Object, String) : void
- ● SetCellOpacity(List<Object>, int) : void
- ● Setup() : void

Appendix 4

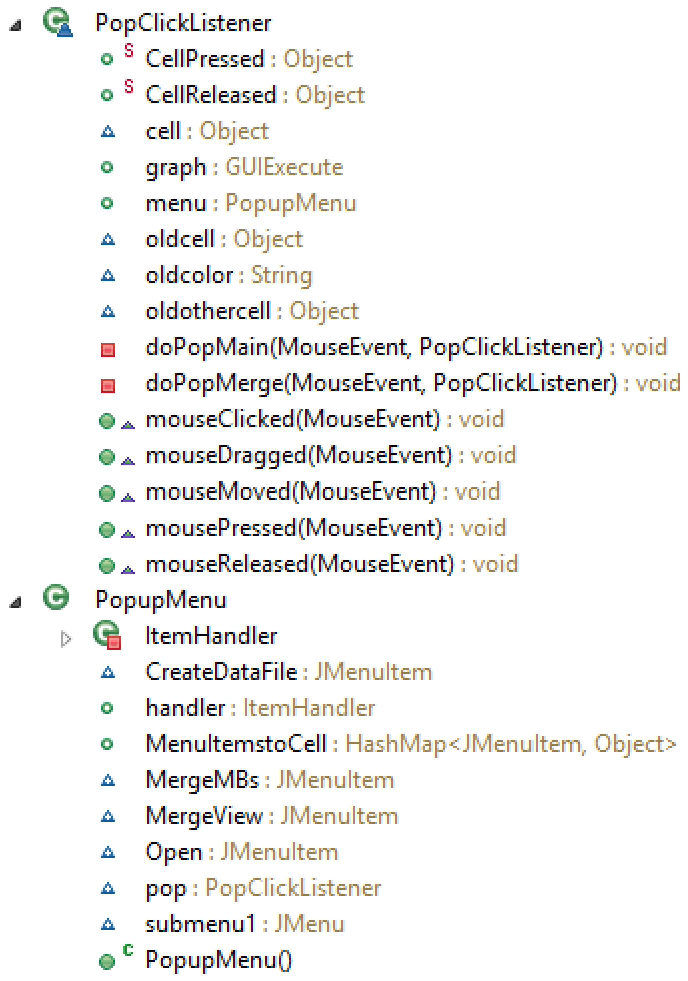
Complete architecture of the `Buttons` class:



```
Buttons
    Actions
    MergeActions
    ActionButtons : List<JButton>
    graph : GUIExecute
    jpanel : JPanel
    MergeButtons : List<JButton>
    actionPerformed(ActionEvent) : void
    CreateMergeButtons() : void
    RemoveMergeButtons() : void
    Setup() : void
```

Appendix 5

Complete architecture of the `PopClickListener and PopupMenu` classes:



```
PopClickListener
    CellPressed : Object
    CellReleased : Object
    cell : Object
    graph : GUIExecute
    menu : PopupMenu
    oldcell : Object
    oldcolor : String
    oldothercell : Object
    doPopMain(MouseEvent, PopClickListener) : void
    doPopMerge(MouseEvent, PopClickListener) : void
    mouseClicked(MouseEvent) : void
    mouseDragged(MouseEvent) : void
    mouseMoved(MouseEvent) : void
    mousePressed(MouseEvent) : void
    mouseReleased(MouseEvent) : void
PopupMenu
    ItemHandler
    CreateDataFile : JMenuItem
    handler : ItemHandler
    MenuItemstoCell : HashMap<JMenuItem, Object>
    MergeMBs : JMenuItem
    MergeView : JMenuItem
    Open : JMenuItem
    pop : PopClickListener
    submenu1 : JMenu
    PopupMenu()
```