D 2026

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

A HOLISTIC APPROACH FOR PARTITIONING AND OPTIMIZING SOFTWARE APPLICATIONS ON FPGAS

TIAGO LASCASAS DOS SANTOS

TESE DE DOUTORAMENTO APRESENTADA
À FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO EM
ENGENHARIA INFORMÁTICA

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A Holistic Approach for Partitioning and Optimizing Software Applications on FPGAs

Tiago Lascasas dos Santos

Doctoral Program in Informatics Engineering

Supervisor: João M. P. Cardoso, PhD

Second Supervisor: João Bispo, PhD

July 31, 2026

A Holistic Approach for Partitioning and Optimizing Software Applications on FPGAs

Tiago Lascasas dos Santos

Doctoral Program in Informatics Engineering

Approved in public examination by the Jury:

President: Prof. Pedro Nuno Ferreira da Rosa da Cruz Diniz
Referee: Prof. Laura Pozzi
Referee: Prof. Carlos Alvarez Martinez
Referee: Prof. Nuno Filipe Valentim Roma
Referee: Prof. João Paulo de Castro Canas Ferreira
Supervisor: Prof. João Manuel Paiva Cardoso

July 31, 2026

Abstract

In heterogeneous computing, a common approach toward improving the performance of applications, in terms of their execution time, is to accelerate critical code regions using FPGA accelerators. This approach often consists of two steps: the first is Hardware/Software partitioning (i.e., the identification of code regions that should be offloaded to an FPGA), and the second is the series of code optimizations applied over the regions that go to the FPGA. Both tasks have been studied in isolation, but there is still a lack of understanding as to how the decisions made in one task can improve the decisions made in the other, and vice versa, so that the whole achieves better performance than the sum of its parts. Our main objective is, therefore, to research a fully automated holistic process of Hardware/Software partitioning and FPGA code optimization where both components complement each other. Through the use of a source-to-source compiler, we build a task-based intermediate representation for large C-language input applications. Then, a greedy algorithm creates clusters of tasks that are suitable for hardware acceleration, taking into consideration an expansion heuristic (e.g., capturing $\approx 80\%$ of the program's CPU execution time) and whether a task should be added to the cluster. To enable a better and more efficient clustering process, we propose source-to-source transformations to tackle non-synthesizable C features, e.g., dynamic memory allocations, system calls, and indirect pointers. Furthermore, we also optimize the memory footprint of an offloaded region using holistic information obtained by analyzing the lifetimes of the data beyond the cluster. Evaluation is done using large applications (i.e., with multiple functions with complex control flow and non-synthesizable C features) from CortexSuite, plus an edge detection application and a *llama2* inference implementation. By creating clusters representing $\approx 80\%$ of the execution time, and by applying all transformations, we achieve region speedups of up to $74.82\times$ when compared to their CPU counterparts, and CPU-FPGA application speedups of up to $8.07\times$.

Resumo

Em computação heterogénea, uma abordagem comum para melhorar o desempenho das aplicações, em termos do seu tempo de execução, consiste em acelerar regiões críticas de código utilizando FPGAs. Esta abordagem consiste, geralmente, em duas etapas: a primeira é a partição Hardware/Software (i.e., a identificação de regiões de código que devem ser transferidas para o FPGA), e a segunda é a série de otimizações de código aplicadas sobre essas regiões. Ambas as tarefas têm sido estudadas de forma separada, mas existe ainda uma falta de compreensão sobre como as decisões tomadas numa tarefa podem melhorar as decisões tomadas na outra, e vice-versa, de modo a que o todo atinja um melhor desempenho do que a soma das suas partes. O nosso principal objetivo é, portanto, investigar um processo holístico completamente automatizado de partição Hardware/Software e otimização de código para FPGAs, onde ambas as componentes se complementam. Através do uso de um compilador fonte-a-fonte, construímos uma representação intermédia baseada em tarefas para grandes aplicações de entrada em linguagem C. De seguida, um algoritmo guloso cria agrupamentos de tarefas adequadas para aceleração em hardware, tendo em consideração uma heurística de expansão (e.g., capturar $\approx 80\%$ do tempo de execução do programa no CPU), e se uma tarefa deve ser adicionada ao agrupamento. Para permitir um processo de agrupamento melhor e mais eficiente, propomos transformações fonte-a-fonte para lidar com características não sintetizáveis da linguagem C, e.g., alocações dinâmicas de memória, chamadas de sistema e apontadores indiretos. Adicionalmente, otimizamos a pegada de memória das regiões utilizando informação holística obtida através da análise do tempo de vida dos dados além do agrupamento. A avaliação é realizada utilizando aplicações de grande dimensão (ou seja, com múltiplas funções, fluxo de controlo complexo e características da linguagem C não sintetizáveis) da CortexSuite, além de uma aplicação de deteção de contornos e uma implementação da inferência do *llama2*. Ao criar agrupamentos que representam $\approx 80\%$ do tempo de execução, e aplicando todas as transformações, alcançamos acelerações de região de até 74,82$\times$ quando comparadas com as suas contrapartes no CPU, e acelerações na aplicação CPU-FPGA de até 8,07$\times$.

Acknowledgments

First and foremost, I thank my advisors, professors João Cardoso and João Bispo, for the unyielding support afforded to me over these last four years. I deeply appreciate everything you have done for me, from your expertise to your mentorship, from your constant feedback to your guidance in keeping me focused, from all the odd hours we faced in rallying towards impending deadlines to the more earnest moments afforded to us from time to time. It has been, once again, an honor and a privilege to work with both of you.

I also extend a particular thank you to Fundação para a Ciência e Tecnologia for their assistance in providing me with the PhD grant 2021.07324.BD, making this whole endeavor possible, and to CMU Portugal, for providing me with the funding and opportunity to visit Carnegie Mellon University for three months.

Everyone involved in SPeCS lab, whether directly or through past affiliation, has proven invaluable to me. If you so much as provided feedback on one of my presentations, developed some tool or software I used, or joined me in a productive conversation, I want you to know that you left a positive impact. In no particular order, I specifically acknowledge João Matos, José Pedro Ferreira, Léo Duarte, Nuno Paulino, Pedro Pinto, and Tiago Carvalho. An even heartier thanks goes out to Luís Sousa, who worked tirelessly on modernizing the in-house technology on which this thesis heavily relies.

A special thank you is also issued to Prof. James Hoe, for having mentored me during my three months at CMU, as well as lending his support to the paper we co-wrote about the results obtained during the visit. I appreciate everything you've done for me, and I look forward to meeting up again someday. Thanks, as well, to Siddharth Sahay and Shashank Obla, for showing me the way around CMU and Pittsburgh, and to João Mesquita and Gabriel Santos for being the best housemates one could ask for.

Last, but certainly not least, I would like to thank my family for their constant, ever-present support. Mom, Dad, I think I can now provide a different answer to the question, "Are you almost done with your thesis?" Thank you for everything you have given me. I also want to thank my brother, Gonçalo, for providing much-needed distractions during those long hours. And, finally, I need to acknowledge my cat Oslo. You came into my life right in the middle of the PhD, and it is no coincidence that everything that came after was that much brighter.

Tiago Lascasas dos Santos

"Any road followed precisely to its end leads precisely nowhere."

Frank Herbert, in *Dune*

Contents

List of Figures

List of Tables

Abbreviations

ACO	Ant Colony Optimization
ASIC	Application-Specific Integrated Circuit
BRAM	Block Random-Access Memory
CC	Cyclomatic Complexity
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DFG	Dataflow Graph
DPR	Dynamic Partial Reconfiguration
DSE	Design Space Exploration
DSL	Domain-Specific Language
DSP	Digital Signal Processor
FF	Flip-Flop
FPGA	Field-Programmable Gate Array
GA	Genetic Algorithm
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLS	High-Level Synthesis
HW	Hardware
II	Initiation Interval
ILP	Integer Linear Programming
IR	Intermediate Representation
LUT	Lookup Table
P&R	Place-and-Route
PL	Programmable Logic
PSO	Particle Swarm Optimization
RAW	Read-after-Write
RTL	Register-Transfer Level
S2S	Source-to-Source
SA	Simulated Annealing
SDRAM	Synchronous Dynamic Random-Access Memory
SoC	System-on-a-Chip
SW	Software
TS	Tabu Search
TU	Translation Unit
URAM	Ultra Random Access Memory
WAR	Write-after-Read
WAW	Write-after-Write

Chapter 1

Introduction

Ever since the end of the Dennard scaling [1], around 2005, CPUs have faced reduced single-threaded performance improvements [2]. Clock frequency could not continue to increase without also increasing the power dissipation of the CPUs, which would itself lead to increased dissipation levels. These challenging increases in heat and power consumption directly contrast with the advent of mobile and edge computing, which often have strict power and thermal dissipation requirements. On the other end of the spectrum, in the realms of high-performance computing, these issues are also present, as workstations, mainframes, and cloud infrastructures become increasingly hard to efficiently scale [3].

The challenges in devising efficient hardware contrast with the ever-increasing need to accelerate software applications, across all modalities: from TinyML [4] AI models and computer vision algorithms running at the edge, to bioinformatics research and time-sensitive financial workloads running in large datacenters, the demand for latency- and energy-efficient hardware that can meet the requirements of modern software has never been higher [5].

In the traditional CPU space, the limitations of single-core were tackled in two major ways. First, single-core performance was improved by further developing and refining superscalar CPU architectures. By exploiting instruction-level parallelism, modern CPU cores are capable of executing dozens of out-of-order instructions at the same time in deep pipelines. While architectural improvements account for most improvements in single-core performance, they still pale when compared to the previous approach of increasing the CPU frequency. Second, CPU designers also shifted toward transitioning from single-core to multi-core architectures, enabling some applications to run faster [6]. However, the same challenges of scalability still exist when looking at each of the cores in isolation, which makes them still unsuitable for applications bound by single-threaded performance. Furthermore, designing and refactoring applications for multi-core is not a trivial task, as exposing thread-level parallelism may be an untenable development effort, and the introduction of parallelism may lead to further issues of concurrency [7].

One other possible solution to this issue, complementary to multi-core CPU architectures, is heterogeneous computing [8, 9]. Heterogeneous computing, in its simplest form, improves the performance of an application by using hardware accelerators to execute performance-critical

regions of the application, while letting the CPU handle the non-critical components, including system calls [10]. These regions are known as "hot regions," "hotspots," or "performance-critical regions," which may be further defined by the performance metrics relative to the application (execution time, energy consumption, or both).

Several hardware accelerators exist, and they are provided in many different form factors and levels of integration: while some may be dedicated devices, others may be embedded in the same fabric as the CPU, giving rise to System-on-a-Chip (SoC) form factors [11]. For instance, the vectorial extensions inside some CPUs, such as Intel's AVX [12], can be considered a hardware accelerator, as they take away control from the conceptually sequential execution of instructions operating over a single register or memory position, and implement efficient parallelized vectorial operations over multiple data units.

The concept of highly tailored hardware capable of performing a single task with utmost performance is better encapsulated in the form of Application-specific Integrated Circuits (ASICs). These are chips that implement a single operation or algorithm with remarkable performance, but at the expense of not being able to do any other operations. This lack of reconfigurability, plus their high manufacturing costs, makes them only suitable for specific use cases (e.g., sound processing, encryption), which are of limited use to most arbitrary applications. Digital Signal Processors (DSPs) answer this limitation by providing some configurability by exposing highly specialized hardware in the form of a small instruction set (i.e., a DSP with Multiply-Add-Accumulate (MAD) instructions can be used to accelerate different noise filtering algorithms). However, they are still specialized hardware that lack both generality (i.e., can be used only for a small subset of problems) and granularity (e.g., a MAD-based DSP focuses on accelerating instructions, rather than entire tasks or even applications).

Highly flexible reconfigurability is, then, of paramount importance for an accelerator to improve the performance of an arbitrary application. Graphics Processing Units (GPUs) are one such accelerator [9]: despite being capable of performing many general computing tasks, they excel at accelerating algorithms with dense matrix and tensor operations. Originally used for 3D graphics rendering, they have quickly expanded to support other workloads, from video encoding to training neural networks. In recent years, we have seen other domain-specific architectures [13] appear, ranging from Tensor Processing Units (TPUs) running training and inference of Large Language Models (LLMs) on datacenters [14], to simpler Neural Processing Units (NPUs) embedded in consumer CPU dies [15].

Those hardware accelerators, however, are still bound by the restrictions identified, as they still ultimately focus on specific high-throughput operations. True reconfigurability comes in the form of Field-programmable Gate Arrays (FPGAs). These are chips that can have their internal functional units rearranged to any given algorithm with the benefits of specialized hardware, while only spending a fraction of the energy required when compared to a CPU execution [16]. FPGAs fit, therefore, the space between the ease of programming a GPU with the extreme performance achieved by an ASIC. Both GPUs and FPGAs are also present at the various levels of hardware, from the embedded space, where they are often coupled with a CPU on a SoC, to the

high-performance computing space, where they come in the form of PCIe cards [17].

1.1 Problem Description

Despite all the advantages that FPGA accelerators bring, they still have some outstanding issues that make their widespread adoption difficult. Workflow A of Figure 1.1 illustrates a typical workflow for accelerating a generic, pre-existing C/C++ application using FPGAs: C/C++ source code serves as input to a High-level Synthesis (HLS) tool [18], which is capable of generating a hardware design that implements the same functionality as the high-level code [19]. These hardware designs are expressed in Hardware Description Languages (HDLs), such as VHDL and Verilog. After the HDL code is generated, it undergoes the processes of Register-Transfer Level (RTL) synthesis and Place-and-Route, outputting a bitstream used to configure the FPGA with the desired functionality [20].

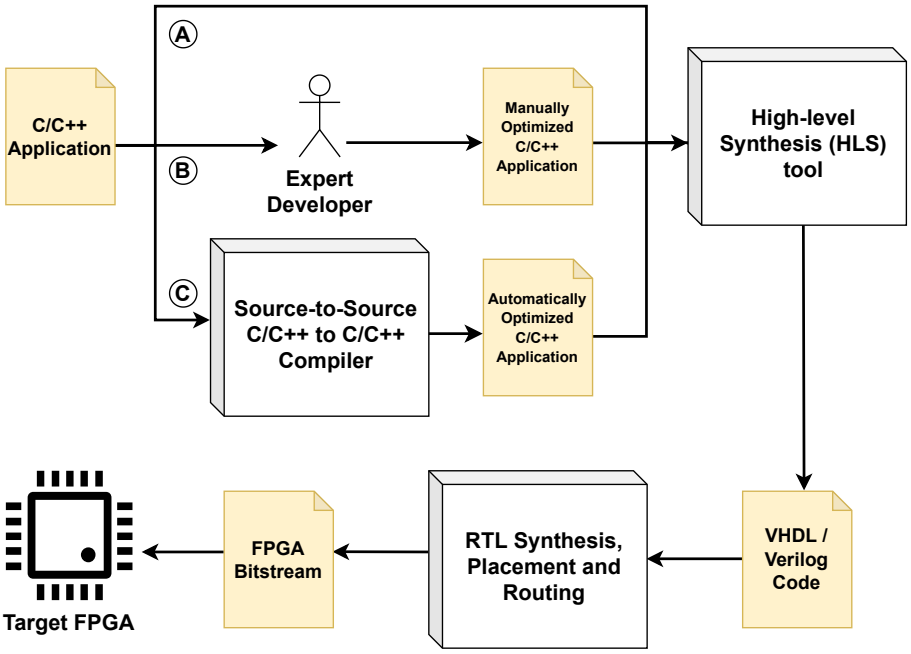


Figure 1.1: Workflows necessary for accelerating a C/C++ application using an FPGA hardware accelerator.

However, this workflow has some issues, as C and C++ source code cannot simply be synthesized as-is. In fact, without applying any transformations, the resultant hardware designs often have performance much lower than the optimal one. This is due to the imperative paradigm followed by C and C++ (most tools only support a subset of C++ with limited use of objects). An imperative paradigm implies that computations are expressed in statements, which execute one after the other, with the possibility of transitioning between code regions through control flow structures. FPGAs, on the other hand, take the most advantage of highly parallel applications, as we may use the hardware's reconfigurability to implement several repeated and parallel structures, at multiple levels of granularity. It is, then, necessary for code to be optimized beforehand,

```
#define N 1024

void functionA(int A[N], int B[N], int C[N])
{
    for (int i = 0; i < N; i++) // Loop 1.1
    {
        C[i] = A[i] + B[i];
    }
}

void functionB(int A[N], C[N])
{
    for (int i = 0; i < N; i++) // Loop 1.1
    {
        int counter = 0;
        for (int j = 0; j < N; j++) // Loop 1.2
        {
            int temp;
            temp = A[i] + C[j];
            temp = temp * temp * 2;
            counter += temp;
        }
        A[i] = counter;
    }
}

// Starting point
void functionC(int A[N], int B[N])
{
    int C[N] = {0};
    functionA(A, B, C);
    functionB(A, C);
}
```

Figure 1.2: Example of an application that can benefit from FPGA acceleration.

on a step before HLS, in order to expose parallelism and thus unlock the true potential of FPGA acceleration [21, 22].

As an example of possible optimizations, let us focus on the code example in Figure 1.2, and consider loop 1 of `functionA`. This loop performs, sequentially, the addition of the elements of input arrays `A` and `B`, storing the result in output array `C`. This algorithm is purely sequential in C, as it would execute the same statement 1024 times, one after the other. On an FPGA, though, we may perform many of these iterations in parallel, particularly since there are no inter- or intra-iteration dependencies. We could even have all 1024 iterations executing in parallel, if the FPGA had enough resources available, with the small caveat that memory throughput would be limited at each clock cycle. That could further be improved by splitting all the arrays into multiple smaller arrays, so that they could all be mapped into different memories and thus reduce or even eliminate the memory bottlenecks. These kinds of transformations would need to be performed either by restructuring the source code, or by specifying directives that tell the HLS tool what to do. Some transformations may be possible to implement using either approach, e.g., loop unrolling, while others may only be possible in one, such as loop pipelining, which is best done by the HLS tool as it often uses a lower-level intermediate representation (IR) closely tied to the scheduling done by the tool for the whole design.

Second, finding the hotspots in an application is not always straightforward: applications often need to be divided into tasks, of which some are selected to go to hardware. Defining what a task is, however, is not trivial, as one can consider different levels of granularity. For instance, in Figure 1.2, we could consider loop bodies as being our tasks, such as the single statement in line 7 and the block in lines $18--21$. Or, we could consider whole loop nests, such as the one consisting of loops 1.1 and 1.2. Further still, we could consider a whole function as a task, or even a set of functions (e.g., we could inline the calls to `functionA` and `functionB` and consider `functionC` as being our task). Determining the tasks, their granularity, and whether or not they should be offloaded to the FPGA accelerator is, therefore, another major challenge, typically addressed as Hardware/Software partitioning [23].

Both these steps are often left as a burden to a developer, as illustrated by workflow B of Figure 1.1. Developers need to be experts in the field, which further increases the barrier to entry to FPGA acceleration through HLS and limits the widespread adoption of this type of accelerator. Automating these processes is, then, necessary for us to make progress toward these targets.

Automation, however, is a complex issue. Workflow C of Figure 1.1 shows an extra step, where the application goes through a C/C++ to C/C++ Source-to-Source compiler and comes out transformed before being passed on to the HLS tool. By using these compilers, we can both study the automatic optimization of source code, be it through code transformations or directive insertion and configuration, and make decisions about which parts of the application should be offloaded to the FPGA.

We have, then, two outstanding issues that require automation. It follows that the state of the art already provides many contributions for each of these issues, but there is still a noticeable lack of research into how both processes - code partitioning and code optimizations - can influence each other. We posit that the next step is to adopt a holistic view and explore the interplay between these processes, as opposed to looking at them independently.

1.2 Objectives

Our main objective is to propose a solution that encompasses both partitioning and optimization, merging them into a single unified and automated process. We believe the interplay between partitioning and optimization should be explored in a holistic sense, using information from the entire application to decide how a given partitioning may affect the optimizations that come after, and how the optimizations can be designed to take advantage of the partitioning results. We call this process Holistic Hardware/Software Partitioning, and our main objective is to lay the foundations for this approach toward both problems.

Furthermore, we want to focus on large C/C++ applications, and move beyond the simple kernels used by most state-of-the-art methods. By leveraging our automated process, we want to accelerate complex applications with multiple functions, written without HLS in mind, and achieve global CPU-FPGA application latency speedups when compared to the execution time of a CPU baseline. This helps to lift the burden of expert knowledge placed on developers who want

to accelerate a pre-existing application using FPGAs, as well as expand HLS toward applications and domains where it is otherwise not in common use.

1.3 Proposed Approach and Contributions

Our proposed solution to these issues is an automatic procedure that partitions applications between a CPU and an FPGA, while also optimizing the portions of the application that are offloaded. We explore the interplay between these two aspects, as the decisions we make in partitioning can influence the kinds of optimizations we can perform later, and the available optimizations may also be improved using information from the partitioning results.

For this purpose, we employ the use of a source-to-source compiler, and focus on the static analysis of source code by analyzing its Abstract Syntax Tree (AST). Our focus is also on large C applications, with multiple functions, as opposed to the small kernels (e.g., PolyBench [24]) often found in the state of the art. Through static analysis, we produce a task-based intermediate representation with variable granularity. Given a large C application, we use a greedy algorithm to cluster tasks together into a large cluster representing the program's hotspot. This expansion is subject to the available resources on the FPGA, the possibility of synthesizing a task through HLS, and the impact the task has on the communication overhead between the CPU and the FPGA. As we focus on the embedded CPU-FPGA space with shared memory between both computation units, this communication overhead is reflected in the memory access times to off-chip memory. As a large application may use more data than the amount available on-chip, the optimizations applied over the cluster focus on heuristics to maximize the amount of data mapped to on-chip memory.

As an example, if we refer back to Figure 1.2, we would begin with a granularity at the level of a loop body, and would increasingly join together blocks until we capture more and more of the functionality in a single cluster. Once we capture both the functionality of functions `functionA` and `functionB`, we unlock a new possible optimization, as this cluster would expose a producer-consumer relationship in array `C`: all positions of array `C` are calculated first, only for them to be used later in another independent calculation. By merging these two flows, we could have some values of `C` produced at the same time as other values of `C` are consumed, therefore increasing the overall parallelism of the application. This optimization could not have been possible if we had, for instance, decided that our granularity was at the function level, and then decided on offloading `functionA` and `functionB` independently to the FPGA (or just one of them, or none; it would depend on the partitioning algorithm being used).

Our proposed approach must be able to answer the following research questions:

- **RQ1**: For a large C application, does offloading increasingly large code regions to an FPGA inherently improve the overall CPU-FPGA application latency, or do the communication burden and synthesis constraints eventually limit the acceleration gains?

- **RQ2**: Can source-to-source transformations tackling non-synthesizable C features enable the creation of larger code regions for offloading?
- **RQ3**: Do optimizations informed by a holistic understanding of the application lead to competitive FPGA execution times when compared to their CPU counterparts?
- **RQ4**: Can our approach be fully automated?

1.4 Thesis Structure

This thesis is organized into 6 chapters. Chapter 1 provides an introduction, starting with the context behind the thesis, objectives, and proposed approach and contributions. Then, we follow up with the problems that arise from that context, which we then formulate as research questions while providing an overview of our proposed solution. Chapter 2 provides the background knowledge behind this thesis, as well as a synthesis of the latest state-of-the-art trends in these fields. Chapter 3 informs about the methodology of the thesis, including the formulation of a task graph representation. Chapter 4 focuses on the clustering algorithm for large C applications, as well as source-to-source transformations that tackle non-synthesizable C language features and their impact on clustering. Chapter 5 builds on the previous chapter by showing how the clusters obtained from the algorithm can be optimized through heuristics and a holistic knowledge of the entire application. Finally, Chapter 6 concludes the thesis and offers some insights regarding future research paths.

Chapter 2

Background and Related Work

In this chapter, we present the background knowledge relevant to this thesis, followed by a summary of the latest state-of-the-art contributions in the fields of Hardware/Software (HW/SW) Partitioning, High-level Synthesis (HLS), and code optimizations and transformations for code targeting FPGAs.

2.1 Background

The process of accelerating an application using an FPGA consists of two main steps. First, we need to analyze the application in order to find which code regions should be offloaded to the accelerator, in a process known as Hardware/Software Partitioning [25, 26]. Then, after selecting those regions, we need to optimize them for the FPGA, as the imperative nature of C and C++ does not lend itself well to the inherent parallelism of those accelerators. In this subsection, we describe these two steps in detail, as well as provide an overview of some recent approaches to these problems.

2.1.1 Hardware/Software Partitioning

In order for an application to improve its performance using a hardware accelerator, we need to first identify the regions of the application that should be offloaded to that accelerator. Profiling the application may help identify hotspots in the application, i.e., regions of code where most of the execution time is spent or energy is consumed. Simply offloading these regions, however, can be a naïve approach, since it seldom accounts for aspects like communication costs or accelerator resources. Therefore, we can achieve a better split between CPU code and accelerator code by modeling the program in terms of tasks, organized in a task graph.

A task is a set of computations that, depending on its granularity, may represent different source code regions, from basic blocks to loop nests to functions, and so on. The granularity of a task is closely tied to the target accelerator, as we may have accelerators that focus on accelerating one specific type of computation (e.g., a Coarse-Grained Reconfigurable Accelerator (CGRA) can be used to accelerate loops [27]), as well as to the complexity of the application itself. Once we

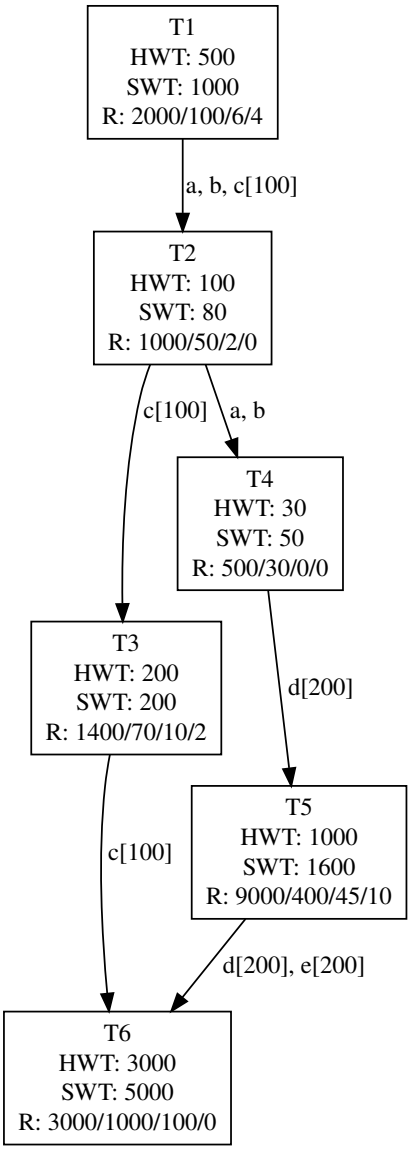


Figure 2.1: Example of a task graph with six tasks. HWT stands for Hardware Time (s), SWT for Software Time (s), and R for Resources (FF/LUT/DSP/BRAM). Edges represent communication between tasks.

decide on a granularity level, each task can be annotated with information about how it would perform if left on the CPU, and how it would perform if it were to be offloaded to the accelerator. The edges of the task graph establish a link between source and target tasks, or between control-dependent tasks. Figure 2.1 shows an example of a task graph with six tasks. Each task is annotated with estimations of execution time and resources in Hardware (in this case, an FPGA), and in Software (i.e., in a CPU). The edges contain information about the data passing between tasks: for instance, task T1 communicates two scalar values, `a` and `b`, plus the array `c[100]`. Naturally, it also follows that T2 can only, by default, begin its execution when T1 finishes, as the edge establishes that relation of precedence.

The communication costs between the tasks can also be reported through the edges of the task graph. These communication costs depend on the architecture of the system, as well as the type of data we are communicating. For instance, scalar values could be passed directly through a channel from the CPU to the accelerator, while bigger data, such as large matrices, could require different approaches: it could be copied to a memory inside the accelerator (e.g., to the onboard Video RAM on a GPU), or it could be accessed through a shared pointer, if both the CPU and the accelerator have access to the same system memory. Therefore, intimate knowledge of the target system is necessary in order to model the communication costs between tasks.

Hardware/Software partitioning is, then, the process of taking a task graph and trying to determine, for each task, whether it goes to the accelerator (hardware) or remains on the CPU (software). With no further constraints, for a task graph with N tasks, we would have a total of 2^N possible solutions to our problem. The number of solutions grows exponentially with the number of tasks, and it is an NP-Complete problem [28]. Solutions are often expressed as a set of bits, with each representing a task. Each bit is set to 1 if the task goes to hardware, or remains at 0 if it stays in software.

In reality, though, we do not need to consider the entire solution space, as we almost always have further constraints and conditions that must be verified in order for a solution to be valid in the first place, such as whether the total set of hardware tasks uses fewer resources than those available in the FPGA. Further constraints exist when we have tasks with instructions that are hard (if not impossible, depending on the architecture) to migrate to an accelerator, such as system calls. The objective function can also be more complex than just trying to minimize the sum of all execution times or energy consumption, as we may also introduce penalties for aspects like communication costs and memory accesses. For instance, in Figure 2.1, task T4 has a better execution time in the FPGA than in the CPU. However, the difference is only 20 s, and when accounting for the communication cost of the two scalar values, it may actually end up taking more time to execute on the FPGA than on the CPU.

The reconfigurability of the accelerator itself may also need to be taken into consideration, as we may have accelerators that use Dynamic Partial Reconfiguration (DPR) [29]. In these systems, instead of having all the tasks configured on the accelerator at the same time, we reconfigure the accelerator each time we want to execute a new task. This has the advantage of reducing the number of accelerator resources needed, as we no longer need to have all hardware tasks configured at the same time, but it also requires us to consider the accelerator reconfiguration time as an additional constraint of the HW/SW partitioning model [30, 31].

Regardless of all the conditions, constraints, and objective functions we have, the problem remains NP-complete. The survey on HW/SW partitioning methods by Hou et al. [28] identifies two main approaches to this problem: the first is to use exact methods in order to find an optimal solution, at the expense of time; the second is to use heuristic methods, such as metaheuristics, in order to reach an acceptable (but not necessarily optimal) solution in a fraction of the time it would take for an exact method to finish.

Exact Methods

In terms of exact methods, Integer Linear Programming (ILP) [32] is by far the most popular one. The problem of HW/SW partitioning is very similar to the 0/1 Knapsack Problem, which is another problem commonly solved with ILP [33]. In this problem, we have multiple items, with different weights, and we want to pack them in a finite number of bags so that we can fit as many items into the bags while maximizing the total weight of each bag. This directly maps to HW/SW partitioning, as our bags would be FPGA resources, our items would be tasks, and our weights would be execution times or energy consumption estimates (with the small difference that in this case we want to find a minimum, and not a maximum).

Other exact methods also exist, such as dynamic programming algorithms [34, 35] and branch-and-bound [36, 37], with the caveat that the latter is also at the basis of many ILP engines.

Heuristic Methods

In contrast to the exact methods, heuristic methods offer far more flexibility and variability of approaches. Of these, we can further identify metaheuristic algorithms as being of particular relevance. Some of these metaheuristics are Genetic Algorithms (GA) [38], Simulated Annealing (SA) [39], Tabu Search (TS) [40], Ant Colony Optimization (ACO) [41], and Particle Swarm Optimization (PSO) [42].

These metaheuristic algorithms can vary considerably, but they mostly operate on the same principle: a baseline solution is first generated, often randomly, to serve as the starting point. Then, a set of neighboring solutions in the solution space is generated and promptly evaluated by a fitness function. This process then repeats, iteratively, for more and more neighboring solutions until an acceptable solution is found. The differences between the different algorithms consist of the number of starting solutions, the way the solution space is generated (e.g., on a bit-based representation, a neighboring solution can be generated by flipping one bit), the criteria by which solutions are accepted or discarded as steps toward finding an acceptable solution, how backtracking in the solution space is handled, and the strategies employed by the algorithm to avoid falling into local maxima and minima.

Outside the metaheuristic algorithms, we may find many alternative approaches such as: a reduction of the problem to Graph Partitioning (much like the problem gets reduced to a 0/1 Knapsack Problem in the exact methods), which can be solved by the Kernighan-Lin heuristic [43]; the use of clustering algorithms [44]; the usage of simple heuristics based on memory access patterns [45]; and Hopfield neural networks [46].

2.1.2 High-level Synthesis

High-level code can be synthesized into a hardware design by the use of High-level Synthesis (HLS) tools [21, 47, 48]. The most common input languages are C, C++, and OpenCL, although we may find HLS tools that accept other languages, such as Java or MATLAB [49]. It is also common for them to accept Domain-Specific Languages (DSLs), particularly in the format of

libraries of abstractions [50]. Regardless, the input source code is almost always from a language with an imperative paradigm, which is reflected in highly sequential programs where statements, executed one at a time, change the program's internal state.

We can find plenty of HLS tools, both commercial and academic. Some attempts have been made at cataloging the several existent tools, with the 2016 survey by Nane et al. [49] being a particularly thorough example. A more recent survey, by Huang et al. [51], focuses on describing the optimizations available on each tool. Out of all the available tools, we have selected a few that we think merit further notice, either by being widely used in the community or by being at the basis of recent and relevant contributions:

- Vitis HLS: previously known as AutoESL and Vivado HLS, it is Xilinx/AMD's proprietary HLS tool [52]. Despite its commercial nature, it is one of the most used HLS tools in academic publications, particularly for those that do not require changes to the tool's internal behavior (although support for user plugins was recently added, allowing for the introduction of new or modified features [53]);

- LegUP: an open-source HLS tool, acquired by Microchip Technologies, that is capable of automatic partitioning of code regions onto an FPGA [54]. It also has a commercial version with an integrated development environment known as SmartHLS [55]. Due to its robustness and open-source nature, it serves as the basis for some contributions that focus on advancing the processes inside the HLS tools [56, 57];

- Bambu: an open-source HLS tool that focuses on modularity, preserving memory semantics, and self-verification of the synthesized results [58, 59]. Its extensibility, in particular, is designed in order to facilitate the research of new HLS techniques and algorithms;

- Dynamatic: a new open-source HLS compiler that converts the input source code into dataflow circuits. These are then scheduled dynamically, as opposed to the typical static scheduling of HLS tools [60];

HLS tools, however, are not typically capable of producing good results by simply taking in the input source code and directly mapping it onto hardware. The imperative nature of the input languages used for HLS conflicts with the highly parallel nature of FPGAs, and simply translating code as-is does not take full advantage of that parallelism. It is, indeed, very likely that the synthesis of unoptimized code may lead to a hardware design that is less efficient than simply executing the program on the host CPU, at least in regard to its execution time.

This limitation can be overcome by introducing several code transformations that expose parallelism in the input program, be it at the operation, loop, or task level. HLS tools can perform many of these optimizations automatically, but some others may need to be specified by a developer, so that the tool knows when to use them. Furthermore, there are also transformations that can be performed outside the HLS tool itself, over the program's source code. Naturally, as we move away the responsibility of optimization from the tool to the developer, we are introducing

hurdles to the development process, but we may also stand to gain so much more than by simply using the optimizations that the HLS tool performs [61, 62].

There are two kinds of optimizations performed by HLS tools. The first category of optimizations consists of those that are performed implicitly, by default, without the developer needing to specify them. These are often simple, such as bitwidth analysis optimization to reduce the hardware footprint; chaining of arithmetic operations to maximize the number of operations per cycle; and speculative code execution, where data is calculated ahead of time, and then accepted or discarded based on an eventual condition [49].

The second category of optimizations present in HLS tools consists of those enabled by directives and compiler flags, thus requiring either human intervention or another program that is capable of reasoning about what design choices to make. Either way, the usage of each of these directives poses several questions: where in the source code should the directive be located; what parameters should be used to configure the directive; how using a directive may change the way other directives perform; and what impact a set of directives would have on the final resource usage of the hardware design.

As an example, let us assume that we want to expose parallelism on a loop nest by using a loop unrolling directive. Regarding the directive's locality, we would need to decide which of the loops we would want to unroll, and which ones we do not. Then, in terms of its parameters, we would need to decide the unrolling factor, and maybe whether to synthesize exit checks for the unrolled loops, if that is supported by the directive. Concurrently to that, we would also need to look at how any other directives being used in the loop could affect the overall design, e.g., if we are also using pipelining directives, then we would need to be mindful of the interplay between these two common loop-based transformations; and, finally, we would need to consider the final resource usage by weighing every single one of the decisions we have made so far, as the size of the loops we are unrolling, the unrolling factor, and the usage of other optimizations would influence the total number of resources needed to implement the loop.

The different transformations and optimizations available in HLS tools have been surveyed by de Fine Licht et al. [63]. They identify three major groups of transformations: pipeline-enabling transformations, scalability transformations, and memory access transformations. Most of these need to be manually specified and configured by directives, as few are performed with no developer input by most HLS tools:

- Pipeline-enabling transformations: these transformations are performed in order to enable better pipelining. Loop pipelining, by itself, is a very common transformation that interleaves the execution of loop iterations in order to exploit the simultaneous execution of operations that have no dependencies, improving operation and loop-level parallelism. Loop pipelining is governed by a parameter called the Initiation Interval (II), which tells us how many cycles are needed in order for the pipeline to begin processing the next iteration of the loop. On a loop with no intra- or inter-iteration dependencies and no memory bottlenecks, the II would be equal to 1, although in practice that seldom happens. Still, achieving the smallest possible II is of paramount importance, and several code transformations can

help the HLS tool achieve adequately small II values. These transformations can range from function inlining, loop flattening/coalescing, and fusing pipelined loops. The authors further identify some implicit transformations performed by the tools themselves, such as buffering data from system memory to on-chip memory;

- Scalability transformations: these transformations focus on improving the performance of applications with high volumes of data by unrolling their loops (which can then be pipelined). They also include loop tiling transformations, as well as introducing dataflow architectures, which is common between a sequence of functions that perform different operations over the same data;
- Memory access transformations: these transformations focus on how to improve the latency cost of accessing memory. Several techniques are offered, from splitting data across multiple, smaller memories, converting to smaller datatypes (with the caveat that program correctness may be compromised), and introducing buffers and queues.

2.1.3 Code Transformations and Optimizations

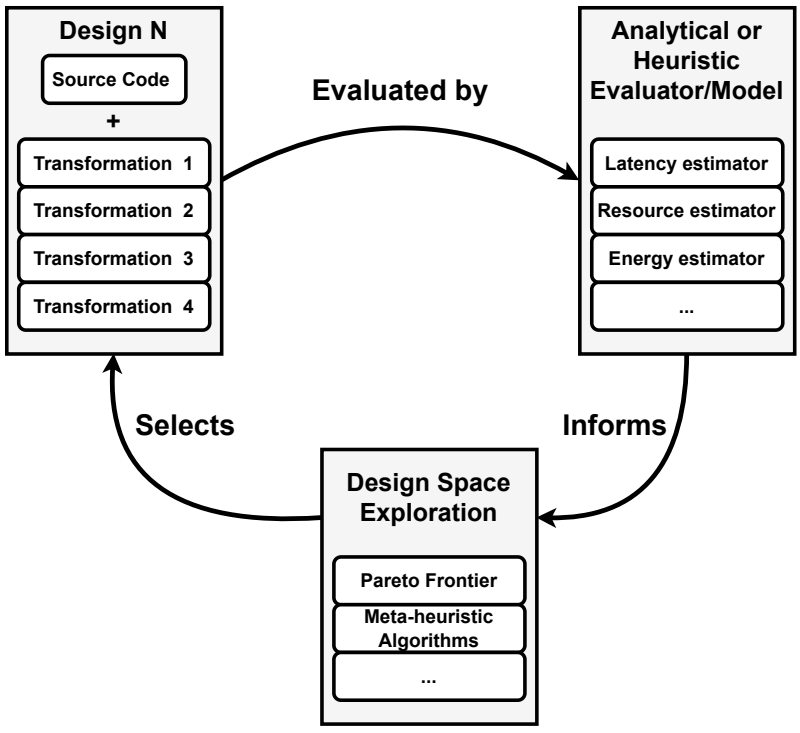


Figure 2.2: Typical optimization flow of an application for HLS: a design space exploration engine selects a design and, based on its estimated performance, it either accepts it as the most optimal design or discards it in favor of a new one, repeating the process until the ideal design is found [51].

As previously mentioned, there is an ever-increasing need to improve the optimization of applications for HLS—both in terms of introducing new optimizations, but also in terms of automating

existing ones. Most importantly, we want to ensure that these processes can be applied to different kinds of applications, as opposed to devising optimization strategies that are tailored to one specific application, such as in [64, 65].

The process of automatic optimization of applications for HLS is summarized in Figure 2.2 and can be roughly split into three tasks [51]: choosing a set of code optimization strategies, often called a recipe, that result in a design when applied over an input source code; evaluating the quality of the design, be it through synthesis or through estimations; and choosing new and better designs using a design exploration engine.

The optimization step performs code transformations in order to improve a C/C++ source code into a representation that is more suitable for synthesis. These transformations can be done on a source-to-source level, be it through the introduction of directives or actual code transformations [66–68], while others may be performed in the intermediate representations inside the HLS tools [60, 69, 70]. Furthermore, the HLS tools may be modified so that certain processes, such as the modulo scheduling of loops and the allocation of operations to DSPs, are more effective [56, 71, 72]. The resulting set of optimizations and transformations applied to the source code or internally by the HLS tool is collectively known as a design.

However, it is often necessary to evaluate the quality of a design. The simplest way to do so is by synthesizing it and obtaining the required metrics through the synthesis report, or by executing the generated bitstream on an FPGA. This is a time intensive process, though, as the synthesis of a single design may take hours to execute. Alternative ways to gauge the performance of a design are then required: through the use of heuristics, analytical models, or machine learning models, we are able to predict, with adequate accuracy, metrics such as the design latency (and therefore its execution time, when provided with a clock frequency), its power consumption and estimated usage of the different FPGA resources. Typical FPGA resources are Flip-Flops (FFs), Lookup Tables (LUTs), Digital Signal Processors (DSPs), and Block Random Access Memories (BRAMs).

Finally, we need to bring these two processes together: now that we know how to generate a design, and how to evaluate that design, we need to iterate through the design space (i.e., the set of all possible designs that can be generated from a given source code) in order to find the one that better suits our performance targets. This process is known as Design Space Exploration (DSE). In DSE, we generate a design, evaluate it and then decide on whether to accept it as our most suitable design, or to discard it and select a new one until the best one is iteratively found. The notion of "best design" is often expressed in terms of Pareto optimality, as we want to select a design where we can no longer improve a metric without making the others come out worse. The process of selecting the next design is often guided by metaheuristic algorithms, such as Genetic Algorithms (GA), Simulated Annealing (SA), and Ant Colony Optimization (ACO) [73]. This is not always the case, though, as exhaustive search of a pruned design space is also possible [74].

It is worth noting, though, that some of these processes may be applied in isolation—code optimization strategies, for instance, may be applied in a way that they only generate one possible design, with the decision process being guided by heuristics or polyhedral analysis [75]. The

estimation step is also often skipped, with the metrics coming instead from simply synthesizing the design [76, 77]. Finally, many contributions focus only on one of these components, as their authors aim to improve a process, or part of a process, without the added entropy of also modifying the others.

2.2 State-of-the-Art

In this section, we summarize the state of the art across four different sub-fields, as well as what major trends and gaps in knowledge exist in each of them. Furthermore, we include an overview of the major benchmarks being used by recent state-of-the-art contributions.

2.2.1 Techniques for Hardware/Software Partitioning

The state of the art in Hardware/Software Partitioning still follows the two previously identified approaches: exact methods and heuristic methods.

In terms of exact methods, the use of Integer Linear Programming is still the most popular approach [78–80]. One notable exception [81] to ILP is through the use of the Balas method, which is itself based on branch-and-bound [82]. We also notice that, in recent years, there has been a shift in the optimization target, with energy consumption becoming more prevalent, as opposed to the execution time of the application [78, 79]. Indeed, multi-objective optimization is also a focus, as we have ILP models that attempt to find designs that minimize both metrics [79].

When it comes to heuristic methods, though, we find more variety of approaches. In terms of metaheuristic algorithms, there are many contributions: some try new metaheuristic algorithms, such as Hybrid Binary Differential Algorithms (HBDE) and Group Theory Optimization Algorithms (GTOA) [83], Firework Algorithms [84], and Binary Firefly Algorithms [85]. However, many of these novel algorithms are not tested with real applications, instead relying on random task graphs generated by TGFF [86].

Simple heuristics, on the other hand, are often applied to real applications. For instance, Wang et al. [87] apply partitioning to an image processing application by using the arithmetic intensity of the functions as their splitting factor (arithmetic intensity being the ratio between the computation and memory accesses of a code region). Prakash et al. [45] go one step further and provide a greedy algorithm that is capable of selecting, with reasonable accuracy, a set of basic blocks that should be implemented in hardware.

Focus, too, should be given to the kind of applications that are the target of partitioning. More specifically, in recent years we start to see neural networks being quite prevalent, with the layers of the network informing the granularity of the partitioning problem [87–89].

Overall, we find that the state of the art is mostly lacking in the evaluation of large applications, particularly regarding the heuristic methods, as the metaheuristic methods focus a lot on artificial data, while the remaining ones tend to focus on specific kinds of applications. These limitations are alleviated by exact methods, although at the expense of higher execution times.

Compiler-Assisted Single-pass Partitioning

Having examined the overall trends in HW/SW Partitioning, we focus on two particular contributions, RegionSeeker [90] and AccelSeeker [91], which more closely match the single-pass approach for clustering tasks for FPGA offloading we propose.

RegionSeeker, proposed by Zacharopoulos et al. [90], performs a single pass through a program's LLVM representation, and finds regions at several granularity levels that are considered good candidates for offloading. This varying granularity level avoids the issues associated with high levels of granularity, such as a function where only some of its parts could benefit from offloading. The process of selecting the regions is enabled by two different algorithms: an exact algorithm based on branch-and-bound, and a greedy (i.e., heuristic) algorithm. They also provide a trade-off between both approaches by using the exact method with a pruned list of inputs.

Building upon this work, the authors subsequently introduced AccelSeeker [91]. Recognizing that intra-function acceleration often fails to amortize the high cost of host-accelerator data transfers over system buses, AccelSeeker operates at the granularity of the LLVM function call graph. To optimize partitioning under area constraints, candidate selection is modeled as a maximum independent set problem on a conflict graph. Unlike RegionSeeker, which strictly prohibits overlapping regions, this alternative formulation permits partial overlaps among candidates by accurately tracking shared function invocations. The optimal subset is then extracted using a recursive algorithm akin to the Bron-Kerbosch algorithm [92].

These approaches align with ours regarding task granularity, the single-pass nature of the clustering algorithm, and the type of input application it considers (i.e., AccelSeeker is evaluated using an implementation of an H.264 decoder [93], which is a complex, multi-function application). However, our approach differs in several key aspects. Firstly, we do not lower the input application down to an IR, like LLVM, retaining the original source code and the potential for retargetability. Secondly, we also consider large applications from multiple domains, including those with non-synthesizable code features, whereas the H.264 decoder used is a clean and synthesizable implementation. Finally, our clustering algorithm takes into consideration code optimizations and transformations applied over the offloaded region, rather than creating raw and unoptimized IR regions requiring further downstream optimization.

2.2.2 Intermediate Representations for Parallel Programs

While we point out that the task graph representation used in most HW/SW partitioning schemes is often sufficient, it cannot be understated how more detailed intermediate code representations can provide better insight into how the code behaves and is organized, thus enriching the analysis that we can do. Several intermediate representations for parallel programs exist, with varying degrees of granularity and targeting a wide gamut of accelerators, rather than just FPGAs (and with a focus on parallelization frameworks, such as OpenMP and Cilk, rather than just simple C and C++).

Most of their use cases fall into two scenarios: improve the parallelism of applications [94, 95], and make it easier for programmers to analyze code [96, 97]. Granularity, however, seems

to remain static throughout most representations, but with some notable exceptions: for instance, Thoman et al. [95] use loop-based tasks, but are then capable of merging them into larger tasks, roughly equivalent to loop unrolling. The Heterogeneous Parallel Virtual Machine (HPVM), by Kotsifakou et al. [94], takes this one step further by representing granularity through a hierarchy, where different hierarchical levels represent different levels of granularity (with the bottom level being single code statements).

2.2.3 Code Optimization Techniques for Efficient High-level Synthesis

There have been many recent advancements in the realm of FPGA code optimizations. Many of these focus on optimizing individual applications, such as Support Vector Machines [74], Neural Networks [65], and Fast Fourier Transforms [64]. However, we are more interested in optimizations that can be applied over a larger variety of applications, rather than those tailored to one single application, or type of application. In this subsection, we present the state-of-the-art in predictive models, design space exploration and code optimizations.

Predictive Models

In terms of predictive methods, we notice some major strides in terms of estimating the performance of individual directives. COMBA, by Zhao et al. [98], is perhaps the most relevant example, where several Vitis HLS directives are modeled using highly accurate mathematical models. The interplay between directives is also taken into account. Perina et al. [99], on the other hand, predict the latency of a design by creating a dataflow graph from profiling information and scheduling it using resource constrained list scheduling. Finally, Choi et al. [100] use results from the scheduling stage of HLS to generate a C-like program that is then used to estimate performance—a process that is considerably faster than just letting the HLS tool continue executing until the design is fully synthesized.

Predictive methods also benefit from modern machine learning techniques, with different approaches using different features for their datasets. For instance, Dai et al. [101] use static data, such as the result from previous synthesis reports, the number of memory accesses and arithmetic operations, target clocks, and so on. O'Neal et al. [102], on the other hand, take a dynamic approach by using values provided by CPU counters obtained during the CPU execution of an application.

We can also find predictive models that model aspects other than latency or energy consumption. Choi et al. [103] identify possible stalls in the hardware design by examining a CPU execution of the application, while Davila-Guzman et al. [104] focus on improving how memory is modeled in prediction models by using static information about loop iterations and the DRAM properties of the target FPGA. We also find some examples of prediction models made for one specific kind of application: Shahshahani et al. [105], for instance, propose a prediction model for convolutional neural networks (CNNs).

Design Space Exploration (DSE)

One major advancement in DSE is leveraging the synthesis results of previous designs in order to decrease the number of iterations required to find an adequate solution. HLS finally has, at this point, enough widespread usage for datasets of synthesized designs to be created. One such example is DB4HLS, by Ferretti et al. [106], which contains over four years worth of synthesis time across 100,000 designs. Ferretti et al. [107] further make use of these designs by designing a Domain-specific Language (DSL) that is able to bridge the gap between designs in the database and a design one wishes to evaluate. They further use this database to train an estimator based on Graph Neural Networks (GNNs) [108], which achieves results comparable to traditional DSE techniques. Wang et al. [73] further improve on this idea of matching a design to existing ones by hashing the design and using Hamming distance to find a similar design in the database. Lastly, GNNs are also used by Prakriya et al. [109] to fine-tune a Large Language Model (LLM) capable of recommending HLS directives.

Other contributions focus on improving the DSE engines themselves. Wang et al. [110] use machine learning to set the parameters of the metaheuristic algorithms used to guide a DSE engine, while Belwal et al. [76] explore the use of Quantile Regression Forests as an alternative to metaheuristic algorithms, which often suffer from scalability issues.

Finally, we also see some approaches that focus on approximating a design to another one whose value is known by having already been explored. Castro-Godinez et al. [111] do this by converting the design to a DFG and then simplifying it, while Xydis et al. [112] model the design space as a time series, which makes it suitable for spectral analysis. This is used to identify hard-to-predict sections of the search space. This information is then used to refine a response surface methodology model that approximates the Pareto frontier.

Code Transformations and Optimizations for Performance

Finally, we focus on the code optimizations for FPGAs. In the state-of-the-art, we still find a large focus on optimizations done by the HLS tool itself. Josipović et al. [60], in their Dynamatic HLS tool, model the source code as a dataflow graph that is then scheduled dynamically, as opposed to statically. This has the advantage of having inherent pipelining, with the added bonus of supporting out-of-order execution of operations. Furthermore, it handles memory dependencies by enforcing queues in the access to memory. Cheng et al. [113] focus, instead, on exploiting the correlation between the latency of operations in a loop body and the dependencies between them, to improve pipelining. This is done by extending the modulo scheduling formulation in the HLS tool with some new constraints. Improving the modulo scheduling formulation is also the focus of de Souza Rosa et al. [56], who reduce the complexity of loop pipelining by reducing it to two optimization problems solvable in polynomial time.

Other novel transformations can also be found in the state-of-the-art. For instance, Chapelle et al. [70] improve the inter-thread communication on an FPGA by safely removing the memory arbiters between them, whenever possible (in this case, a thread can be assumed to be equivalent

to multiple instances of the same function executing at the same time). They do this by employing a formal verification tool; Chowdhury et al. [66] find variables that change very little throughout the program execution, and change them to alias with other similar variables or convert them to a constant; Cong et al. [67] propose Latte, a library of pipelined transfer controllers capable of efficiently implementing communication patterns across functional blocks, such as scatter, gather, broadcast, and reduce patterns; Dewald et al. [114] insert checks to determine whether the iterations of a loop with ambiguous intra- and inter-iteration dependencies are independent; Liu et al. [75] explore loop splitting using polyhedral analysis to improve pipelining; and finally Ferreira and Cardoso [115] and Campos and Cardoso [116] perform folding/unfolding operations over a dataflow graph, obtained through code instrumentation, to generate highly parallel C code suitable for FPGAs.

As for BRAM management in FPGAs, we highlight the work of Cong et al. [117], which focuses on techniques for restructuring BRAM buffers to improve bandwidth. While they are applied to small applications from MachSuite [118] and already assume all DRAM buffers map to BRAM, they are nevertheless relevant in the context of a larger region, and could inform a future version of our heuristics. Dessouky et al. [119] offer a different but complementary insight, as they tackle the scarcity of BRAMs, and how dynamic BRAM management can avoid the resources wasted by worst-case (i.e., maximum size) allocations. However, it is not an HLS-based approach, and is difficult to transpose to the context of a large synthesizable C code region. Finally, Vasiljevic and Chow [120] propose using buffer packing, i.e., joining multiple buffers into a single buffer to reduce the amount of unused space in BRAMs.

Overcoming Non-synthesizable C-language Features

Contributions to enabling synthesis in the presence of non-synthesizable language features tend to focus on specific limitations, rather than offering methods that address multiple restrictions at once. Some of these focus on dynamic memory allocation, such as Hi-DMM [121] and *libmem* [122]. These solutions provide custom allocators for BRAM-mapped regions and do not consider memory mapped to DRAM. Thomas et al. [123] offer a specialization of this to dynamic data structures, and while DRAM is now considered, it is limited only to this scenario. The other major set of contributions offered by the state-of-the-art focuses on recursion [123–125], except for HeteroRefactor, a tool proposed by Lau et al. [126]. While this tool still focuses predominantly on removing recursion and transforming data types, it also performs some transformations to remove non-synthesizable pointer and `struct` features, but only applied to small programs and with no major focus on how they may enable the offloading of regions from large applications.

We also highlight some work on performing host calls (e.g., `printf()`) from an FPGA. One approach toward enabling these types of calls is to frame the problem in terms of Remote Procedure Calls (RPC) methods for FPGAs. One such method is *rpcfpga*, proposed by D'Hollander et al. [127], which focuses on communicating fixed-length arrays across a network of FPGAs and a CPU. Similarly, Fleming et al. [128] propose *PushPush*, which contrasts with *rpcfpga* by allowing

the remote execution of any function, regardless of its data types. These two approaches, however, implement this communication mechanism through low-level AXI interfaces and custom IP blocks, and are ultimately hardware-centric approaches hard to leverage in a software-centric environment relying purely on HLS. The same applies to Sumeet and Nambiar's *HLS_PRINT* [129], which allows for `printf()` calls in HLS by coupling an IP block to the kernel.

In addition, we note the *OmpSs@FPGA* programming model proposed by Bosch et al. [130]. In this model, both CPU and FPGA tasks are managed by a task scheduler, allowing for an FPGA task to halt its execution for a CPU task to execute, and vice versa. *OmpSs@FPGA* works well for task-driven applications written with that model in mind, but can be more difficult to adapt to applications without a clear task structure due to the associated programming effort.

Automatic Optimization Frameworks and Tools

The approaches previously described focus on providing individual transformations, often applied to specific applications. A long-standing goal in the community is researching automatic frameworks and tools capable of applying transformation recipes over an input application, raising the abstraction level and easing the adoption of HLS-based flows.

Of particular interest are source-to-source compilers, which transform the code before synthesis. ScaleHLS, by Ye et al. [131], is built using the Multi-level Intermediate Representation (MLIR) [132]. This representation allows for a hierarchical approach to code optimization by providing different IR "dialects", each of which is suited to one specific kind of analysis. The three main optimization levels are the HLS directives, graph transformations and loop-based transformations. MLIR is currently informing further research into code optimizations, with some work-in-progress [133] research trying to use it to improve the polyhedral analysis of loop nests. Furthermore, Pouget et al. [134] bring forth Prometheus, a framework that combines several optimizations (e.g., task fusion, loop permutation) in a single search space, which is searched through using Non-Linear Programming.

It is worth noting that while both ScaleHLS and Prometheus have outstanding quality of results in terms of achieved speedups, they both focus predominantly on small, single-function compute kernels (e.g., from PolyBench [24]), and remain untested under large code regions.

Overall Trends

In general terms, code optimizations for FPGAs still fall short in terms of applying to a wide gamut of applications. Many transformations are specific to an application, or family of applications, and remain unproven when dealing with applications outside their domain. The same still holds for the frameworks and tools capable of automatically applying several transformations to input applications, as while they prove that top-to-bottom automation is possible with good quality of results, they still fail to scale beyond simple kernels.

The latest advancements in design reuse for predictive models and DSE, however, are enticing, and the advent of MLIR has untapped potential, particularly for optimizations performed on source-to-source compilers outside the HLS tool itself.

2.2.4 Combining Partitioning with Code Optimizations

Table 2.1: Benchmark suites from the state-of-the-art

Suite	Language/interface	#Apps	Description	Ref.
AxBench	C++	7	Approximate Computing benchmarks	[135]
CHStone	C	12	Kernels for HLS	[136]
CortexSuite-vision	C	9	Large Computer Vision applications	[137]
HiFlipVX	C++	1	Image processing library for HLS	[138]
llama2.c	C	1	Large Language Model inference	[139]
MachSuite	C	19	Low-level kernels for HLS	[118]
Parboil	C, OpenCL, CUDA, MPI	11	Large applications for CPU-GPU comparisons	[140]
Parsec	C, OpenMP	13	Multithreaded applications	[141]
Perfect	C	15	Kernels from multiple domains	[142]
PolyBench	C	30	Kernels with static loop iterations	[24]
Rosetta	C++	6	Large applications for HLS	[143]

To the best of our knowledge, there are no contributions yet that specifically focus on looking at HW/SW partitioning and code optimizations as a single problem. However, we highlight two relevant contributions that take some steps in that direction: the first is by Zuo et al. [80], who complement their ILP-based partitioning formulation with the random selection of a code optimization recipe from a small set of possible optimizations.

The second is by Führ et al. [78], who combine several configurations of loop unrolling, pipelining, and array partitioning together with several candidate regions for offloading. These are then provided as the input to an ILP-based formulation that minimizes energy usage.

In both processes, there is no back-and-forth synergy between them, as all the decisions about possible optimizations have already been determined before partitioning (e.g., the results of partitioning could never lead to the possibility of previously unconsidered optimizations being applied). Furthermore, both contributions focus on small kernels, rather than applications with multiple functions, which is also a limitation our work seeks to overcome.

2.2.5 Benchmarks

Finally, we include a list of benchmarks commonly used in the state of the art. We have organized these in Table 2.1, where we provide their source languages, number of applications, and a succinct descriptor. Some of these, such as Rosetta [143] and HiFlipVX [138], have not yet seen widespread use for HLS, but we include them regardless, as they have been specifically designed for FPGA evaluation.

Many of these benchmarks also focus on applications with small kernels, or with a reduced number of functions. Benchmark suites like PolyBench [24], CHStone [136], Perfect [142], and MachSuite [118], while popular for validating HLS workflows, are notorious examples of this.

Other suites offer larger applications, such as Parboil [140] and Parsec [141], but are CPU- and GPU-focused applications with generally well-defined regions for acceleration, making them less interesting in the context of HW/SW partitioning.

One suite, CortexSuite [137], stands out by having large and complex applications with no obvious cut-off points for offloading, while also relying on system calls and dynamic memory allocations, as they were not implemented with HLS as a possible target. More specifically, the *vision* subset of CortexSuite contains computer vision applications that could benefit greatly from FPGA acceleration.

2.3 Summary

In this chapter, we started by providing the background in HW/SW partitioning, High-level Synthesis, and code optimizations targeting FPGAs, followed by a state-of-the-art survey of recent developments in these fields. These included advancements in HLS tool efficacy, intermediate representations for programs undergoing HW/SW partitioning, and code transformations and optimizations. The latter focused on transformations from a source-to-source domain, ranging from architectural changes to expose parallelism, dealing with non-synthesizable C-language features, and automatic application optimization frameworks and tools.

Overall, we identify some tendencies in the state of the art. Firstly, we note the general lack of approaches combining both partitioning and optimization, with the notable exceptions of Zuo et al. [80] and Führ et al. [78]. Secondly, partitioning algorithms, and the heuristic ones in particular, still rely predominantly on synthetic task graphs, and offer few solutions as to how a task graph can be generated from any input application. Finally, when partitioning algorithms are validated using real applications, they are often small, single-function applications, with the granularity of the partitioning problem centering around loops. This is a commonality with the HLS optimization tools and frameworks, which also tend to focus on simple applications.

Therefore, we identify gaps in knowledge regarding the lack of a unified approach, the lack of a representation suitable for any input application, and the lack of validation using large applications, which our proposed approach aims to fill.

Chapter 3

Methodology

In this chapter, we start by discussing the approach we adopt for our problem: how we analyze an input application, which task graph formulation we should use, how we can generate task graphs for any input application, and how they can be used in a holistic HW/SW partitioning method. We then describe our evaluation strategy, including the target platform, benchmarks, and the metrics of the evaluation. Finally, we include a motivational example to illustrate the concepts presented herein.

3.1 Leveraging Source-to-Source Compilation and HLS

In order to provide an approach capable of dealing with complex, generic applications, we leverage source-to-source compilers. By leveraging the Abstract Syntax Tree (AST), we perform analysis, transformations, and optimizations, and output the modified source code. This allows for better retargetability, as it is agnostic to any other downstream tools (e.g., HLS tools, C/C++ compiler, static code analysis tools), while also preserving a measure of code readability.

In the context of HLS, source-to-source compilers enable transformations at multiple levels. For instance, we can transform the application at an architectural level to expose task-based parallelism and data streaming, leveraging information and language constructs not available at lower-level intermediate representation (IR); we can also complement these code transformations with HLS directives, which guide the HLS tool to further transform the source code (e.g., unroll loops) and provide tailored information, e.g., regarding how arrays should be mapped and partitioned, and how loops should undergo pipelining.

More concretely, we settle on the Clava C/C++ to C/C++ source-to-source compiler [144] as the basis for building our partitioning and optimization process. Clava enables a developer to programmatically analyze, transform, and optimize source code at the AST level, and has already achieved success as the basis for an HLS code optimization framework [145].

Clava's AST is obtained from *clang*, and it allows a developer to manipulate the AST by using the LARA framework [146]. By providing a powerful JavaScript API, LARA allows, at its most basic level, CRUD (Create, Read, Update, and Delete) operations on AST nodes, as well as

enabling advanced querying over those nodes (e.g., selecting all the nodes that represent the innermost loops in a loop nest). Furthermore, LARA provides general-use transformations and analysis passes, e.g., automatically inserting timers around code regions for instrumentation, or generating Control-flow Graphs (CFGs) for the current AST. As LARA uses Node.js as its JavaScript runtime, we can also leverage TypeScript (i.e., a compile-time type system for JavaScript) to build, distribute, and combine complex extensions for the compiler. This enables higher developer productivity when compared to traditional C and C++ compiler plug-ins found in *gcc* and *clang*. After applying transformations to the AST, Clava uses it to generate and output readable C and C++ code, making it suitable as the first stage in complex compilation flows.

In terms of HLS tools, we settle for AMD/Xilinx's Vitis HLS, as it is the most common HLS tool used by the contributions in our State-of-the-Art survey. Vitis HLS operates in two modes: synthesis and implementation. Synthesis mode generates the HDL design for the input C/C++ source code, in Verilog and VHDL. It also provides us with an estimate of the latency and resource usage of the eventual bitstream implementation. As synthesis is fast (i.e., on the order of minutes even for large and heavily pipelined designs), we can use its estimates to help guide our process. Implementation mode, on the other hand, performs RTL synthesis and Place-and-Route (P&R) of the HDL model, providing us with the final bitstream, real resource usage, and the maximum FPGA clock frequency.

We summarize the entire toolchain for our approach in Figure 3.1. At a conceptual level, our approach consists of five stages, expressed as library-like plug-ins to the Clava compiler:

1. **Subset reduction**: the first step aims to alleviate the entropy associated with large C/C++ applications by reducing the code down to a well-defined subset of C. This is done primarily through source-to-source transformations, e.g., function outlining and constant folding/propagation;

2. **Task Graph generation**: the second step creates a task graph representation of the application, which is used to guide the partitioning process. More specifically, we propose a new task graph representation, the Extended Task Graph (ETG), capable of handling different levels of granularity and application complexity;

3. **Holistic partitioning algorithm**: the third step applies a single-pass greedy clustering algorithm to the ETG, to create a cluster of tasks for offloading according to some criteria. This algorithm is enriched by information obtained through profiling the input application using, e.g., *gprof* [147], as well as by the HLS synthesis estimates for each of the ETG's tasks.

4. **Transformations on the cluster**: the fourth step applies several code transformations on the code region matching the selected cluster for offloading. These transformations range from removing non-synthesizable C-language features to reducing the out-of-chip memory accesses of the offloaded region. We also use *Frama-C*, a static code analysis tool, to obtain metrics about the code complexity of the region;

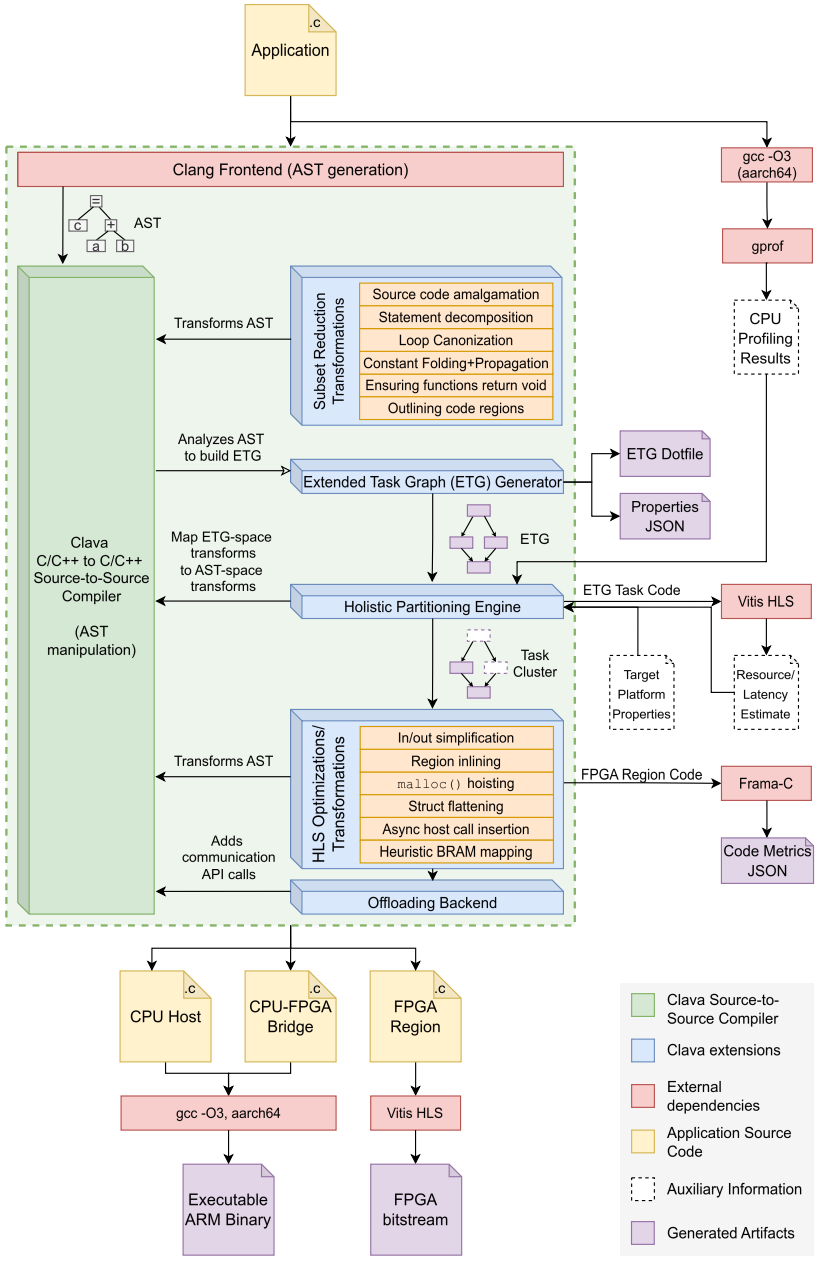


Figure 3.1: Overview of the toolchain for automating the holistic code partitioning and optimization process.

5. **Offloading backend**: the fifth and final step builds all the necessary communication boilerplate between the host and device code. We specifically use the Xilinx Runtime (XRT) [148] for this purpose, although the approach is generic enough to support other backends, such as OmpSs@FPGA [130] or conventional OpenCL [149];

We further note that the specialization of our toolchain into FPGA-based HW/SW partitioning with Vitis and XRT as the target is only a matter of scope, and that all components can easily be adapted to other scenarios. For instance, the ETG representation is independent of any FPGA

particularities, and the holistic partitioning algorithm can be tailored to support other accelerators, such as GPUs or DSPs.

3.2 Reducing the AST to a C Subset

While a source-to-source approach for C/C++ comes with the advantages of preserving retargetability and readability, it also comes with an increased level of complexity. AST-based analysis and transformations are more onerous to apply due to the increased entropy associated with source code, when contrasted to the simpler constructs and assurances of an IR (e.g., static single assignments (SSA) forms). High entropy is reflected in the variability of constructs and language patterns, e.g., a memory position can be accessed in different but equivalent ways (`*buf[5]`, `buf[5][0]`, `**(buf + 5)`). This variability in syntax is further exacerbated in the presence of large C/C++ applications.

Our solution is to apply a series of code transformations that help reduce the entropy of an input source code, in order to reduce it down to a well-defined subset. Our transformations apply to both C and C++, although the latter can only be reduced if it follows a predominantly imperative, C-like coding style (i.e., does not use classes, templates, anonymous functions, or other features introduced in C++11 and higher). We divide these transformations into two recipes: statement-level transformations that lower the entropy of the source code, and function/call-level transformations that expose a task structure to facilitate the posterior generation of a task graph. The first transformation recipe consists of:

1. **Source code amalgamation**: despite Clava allowing for analysis and transformations across different translation units (TUs) (i.e., different `.c` files), we amalgamate all translation units into a single TU. This allows us to simplify access to global variables, as we no longer rely on `extern` keywords. Furthermore, it also facilitates the removal of functions that are never called as we can ensure that the program's function call graph spans a single TU. Variables and functions defined with the `static` keyword, i.e., local only to the TU they were declared in, are renamed in case of symbol conflicts;

2. **Statement decomposition**: we select some transformations proposed by Matos et al. [150] to rearrange variable declarations and expressions to always follow these assumptions:

    - Only one variable must be declared per statement (i.e., transform `int x, y;` into `int x; int y;`);
    - All variable declarations must be at the beginning of their respective scope;
    - Variable declarations must either be uninitialized or have a constant initialization. Any initializations using more complex expressions are moved to separate statements;
    - A function call with a return value must always be in a separate statement, assigning its return value to a variable. More concretely, if a function call is part of an arithmetic expression (e.g., `a = b + foo(c);`), or part of another function call (e.g.,

`bar(foo(c));`), it must be hoisted into a separate statement assigning to a temporary variable, which is then used in the expression (e.g., `int tmp = foo(c); a = b + tmp;`);

- The arguments of a function call must be either simple references to other variables, or references surrounded by unary pointer operators (e.g., `foo(a,&b, *c);`). Any other expressions must be hoisted into a temporary variable;

3. **Loop normalization**: all `for-loops` must have a complete loop header with one loop counter declaration, exit condition, and increment operation. Any loops that do not follow this format, or whose conversion is not possible, must be transformed into `while-loops`;

4. **Array flattening**: all N-dimensional arrays are flattened into 1-dimensional arrays, within the limits of what is possible using static code analysis. This means changing not only the declaration of these arrays, but also the subscript of every array access. This conversion is done for practical reasons, as HLS tools often work better with $1-$dimensional arrays, and also because downstream CPU-FPGA communication APIs, such as the aforementioned XRT, are not compatible with dimensions higher than 1. However, we do keep note of the original access, as that may provide us with additional information when it comes to analyzing memory patterns;

5. **Constant folding and propagation**: while these two transformations are often applied as performance optimizations by downstream compilers and HLS tools, we apply them at this early stage in order to further simplify the source code.

Except for source code amalgamation, all transformations in the recipe are applied multiple times until we arrive at a fixed point, where no more changes occur in the code, as they may positively influence each other (e.g., an arithmetic expression can be simplified down to a single variable or constant, allowing it to be passed directly as a call argument). Listing 3.2 shows a function that undergoes all the transformations in this recipe.

After applying the first subset reduction recipe, we can generate a call graph. In this graph, each vertex represents a function, and edges represent a function call, e.g., an edge $A \rightarrow B$ means function A has a call to function B. Figure 3.3 provides an example of a function call graph. We further annotate the edges of this graph with information regarding the function arguments that are used in the call. We distinguish pass-by-reference arguments that get modified inside the called function from the other types of arguments (read-only pass-by-reference arguments and pass-by-value arguments). We also need to consider the scenario where the same function call happens multiple times inside the caller (e.g., when it is inside a loop). We solve this by adding a number of iterations to the edge that represents the function call, as long as we can statically infer that value.

Finally, there is another important distinction that our function call graph has when compared to those commonly found in the literature: different call sites map to different function nodes, e.g., if function A is called from two different call sites, in functions B and C, we have edges $B \rightarrow A_1$

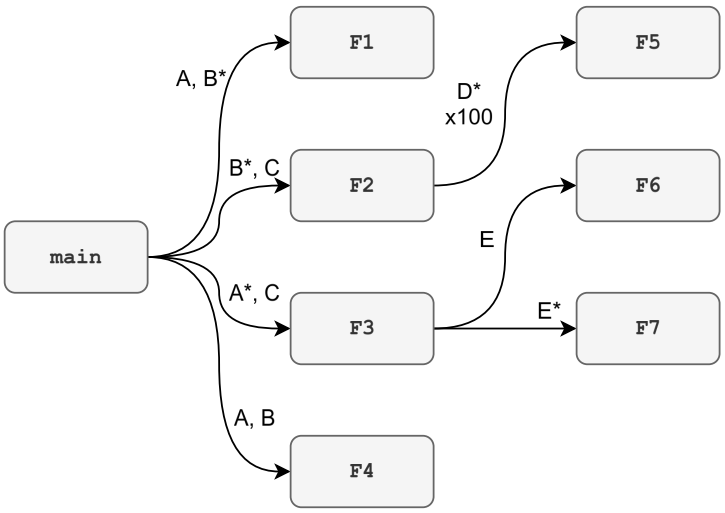
```c
// Before first subset reduction recipe
int main() {
    int a = 2;
    int b[100] = {0};
    char c[100][100] = {{0}};
    int x, y;

    x = foo(a, b);
    y = bar(foo(x, b), x + 1);
    c[2][4] = a * x;

    int acc = 0;
    for (; x > 0;) {
        acc += y * b[x];
        x--;
    }
    return 0;
}

// After first subset reduction recipe
int main() {
    int a = 2;
    int b[100] = {0};
    char c[10000] = {0};  // array flattening
    int x;
    int y;                // one declaration per line
    int acc = 0;          // moved declaration
    int tmp1;
    int tmp2;

    x = foo(2, b);        // constant propagation
    tmp1 = foo(x, b);     // call hoisting
    tmp2 = x + 1;         // expression hoisting
    y = bar(tmp1, tmp2);
    c[204] = 2 * x;       // array flattening

    while (x > 0) {       // loop normalization
        acc = y * b[x];
        x--;
    }
    return 0;
}
```

Figure 3.2: Example of a C program, before and after applying the first subset reduction recipe.

and $C \rightarrow A_2$. This is valid even for call sites within the same function, e.g., $B \rightarrow A_1$ and $B \rightarrow A_2$. The subset reduction transformations facilitate this mapping by associating each call site with a unique line number. In practice, this function call graph is in reality a function call tree, as each node, except for the root, has exactly one parent.

The second set of transformations completes the subset reduction preprocessing step. These transformations are focused on functions and their calls, and rely on the function call tree. The purpose is to ensure that every function in the program falls into one of two categories:

1. **Type A functions**: these functions contain no calls to other functions. An explicit exception is made for calls to functions from `math.h`. These functions can have side effects, including modifying global variables;




```
void F1(int *A, int *B) {
    B[0] = A[0];
}

void F2(int *B, int *C) {
    int foo = C[0] + C[1] + C[2];
    int D[100] = {4444};
    for (int i = 0; i < 100; i++)
        F5(D);
    B[0] = D[0] + foo;
}

void F3(int *A, int *C) {
    int bar; = A[1] * C[1];
    int E[100] = {5555};
    int fizz = F6(E);
    F7(E);
    A[0] = bar + fizz + E[1];
}
```

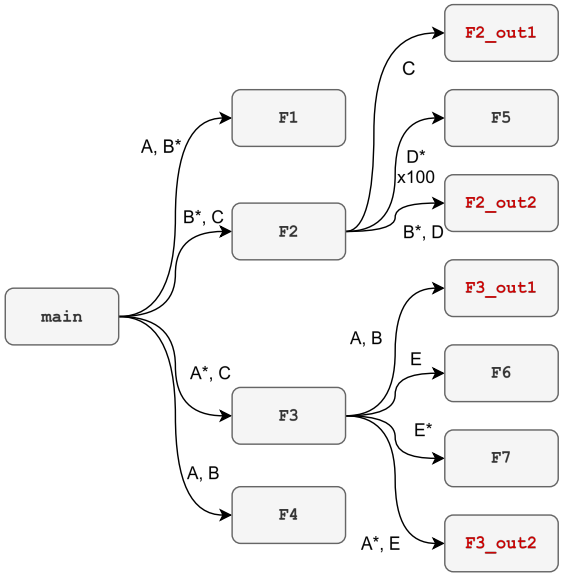
```
int F4(int *A, int *B) {
    return A[0] + B[1];
}

void F5(int *D) {
    D[0] = D[1] * D[2];
}

int F6(int *E) {
    return E[0] + E[1];
}

int F7(int *E) {
    E[0] = E[1] * E[4] * 42;
}
```

```
int main() {
    int A[100] = {1111};
    int B[100] = {2222};
    int C[100] = {3333};

    F1(A, B);
    F2(B, C);
    F3(A, C);
    int res = F4(A, B);
    res += res + 15;
    return res;
}
```

Figure 3.3: Example of a function call graph. Vertices are functions, and the edges represent a function call from the source (caller) to the target (callee). Edges are further labeled with the arguments of the call, with an asterisk (*) representing pass-by-reference arguments that are modified by the callee.

2. **Type B functions**: these functions only contain calls to other functions. Two shallow control flow structures are allowed: loops with a single level of nesting whose body consists of function calls, and if/else blocks with only function calls for either alternative. As assignments are not allowed, all function calls must not have return values, except for `main()` and functions whose implementation is not available (e.g., calls to library functions).

Figure 3.4 shows the call graph from Figure 3.3 after undergoing these transformations. Furthermore, each call site must uniquely map to a function, so that no two call sites can map to the same implementation. These conditions complete the subset, and are necessary for generating a task graph. However, as most applications do not follow this format, we need to apply the second recipe to ensure the application falls into this format:




```
void F1(int *A, int *B){
    B[0] = A[0];
}

void F2(int *B, int *C){
    int foo;
    int D[100] = {4444};
    F2_out1(C, &foo);
    for (int i = 0; i < 100; i++)
        F5(D);
    F2_out2(B, D, foo);
}

void F2_out1(int *C, int *foo){
    *foo = C[0] + C[1] + C[2];
}

void F2_out2(int *B, int *D,
                     int foo){
    B[0] = D[0] + foo;
}
```

```
void F3(int *A, int *C){
    int bar;
    int fizz;
    int E[100] = {5555};
    F3_out1(A, B, &bar);
    F6(E, &fizz);
    F7(E);
    F3_out2(A, E, bar, fizz);
}

void F3_out1(int *A, int *B, int *bar){
    *bar = A[1] * C[1];
}

void F3_out2(int *A, int *E, int bar, int fizz){
    A[0] = bar + fizz + E[1];
}

int F4(int *A, int *B){
    return A[0] + B[1];
}

void F5(int *D){
    D[0] = D[1] * D[2];
}
```

```
int F6(int *E, int *fizz){
    *fizz = E[0] + E[1];
}

int F7(int *E){
    E[0] = E[1] * E[4] * 42;
}

int main(){
    int A[100] = {1111};
    int B[100] = {2222};
    int C[100] = {3333};

    F1(A, B);
    F2(B, C);
    F3(A, C);
    int res = F4(A, B);
    res += res + 15;
    return res;
}
```

Figure 3.4: Example of a function call graph, after lowering every function under `main()` to a C subset. Vertices are functions, and the edges represent a function call from the source (caller) to the target (callee). Edges are further labeled with the arguments of the call, with an asterisk (*) representing pass-by-reference arguments that are modified by the callee.

- **Function outlining**: extracts a code region into a new function, and replaces the region with a call to the new function. It can be applied to any code region, as long as its boundaries are at the same scope level. The compiler iteratively and exhaustively outlines as many code regions as possible, using pre-existing function calls and control flow structures as the boundaries;

- **Function voidification**: transforms each function with a return value to return `void` instead. The return value is instead stored on an additional pointer argument, e.g., `int res = foo(a, b)` becomes `int res; foo(a, b, &res);`;

```c
void outlined_fun_1(int x, int *tmp2) {
    *tmp2 = x + 1;
}

void outlined_fun_2(int *c, int x) {
    c[204] = 2 * x;
}

void outlined_fun_3(int y, int *b, int *acc, int *x) {
    *acc = y * b[x];
  (*x)--;
}

void foo_replica1(int n, int *b, int *res) {
    // body of original foo()
}

void foo_replica2(int n, int *b, int *res) {
    // body of original foo()
}

int main() {
    int a = 2;                  // variable declarations allowed
    int b[100] = {0};
    char c[10000] = {0};
    int x;
    int y;
    int acc = 0;
    int tmp1;
    int tmp2;

    foo_replica1(2, b, &x);     // call to a replica of foo
    foo_replica2(x, b, &tmp1); // call to another replica
    outlined_fun_1(x, &tmp2);  // statement outlined to a function
    bar(tmp1, tmp2, &y);        // voidified return
    outlined_fun_2(c, x);       // statement outlined to a function

    while (x > 0) {                       // shallow loop allowed
        outlined_fun_3(y, b, &acc, &x);  // loop body outlined to a function
    }
    return 0;                   // main() is an exception to voidification
}
```

Figure 3.5: Example of a C application after applying the second subset reduction recipe.

- **Function replication**: by using the function call tree, we find all call sites that point to the same function, and create unique instances of that function for each call site.

Figure 3.5 shows an example of these transformations applied to a program with multiple functions, using the result from Figure 3.2 as the starting point. One of the advantages of function replication is already apparent in this example, as the call to `foo_replica1` has a constant argument. This can lead to the specialization of this instance, as one of the parameters is known at compile-time.

3.3 The Extended Task Graph (ETG) Representation

Having settled on a well-defined subset of C, we can now generate a task graph for a given input application. As we need a task graph definition applicable to any application, we use the assur-

ances provided by the subset regarding Type A and B functions as the basis for our definition of task. For an input source code in this format, we propose the Extended Task Graph (ETG) representation:

Extended Task Graph definition: the ETG is a **Directed Acyclical Multigraph**, defined by a tuple G consisting of a task set T and a communication edge set E.

$$G = \langle T, E \rangle$$

T consists of three sets, each containing vertices of one specific type. TR contains tasks of the type "*REGULAR*"; TE contains tasks of the type "*EXTERNAL*"; and TM contains three tasks, representing the source, sink, and a provider of global variables.

$$T = \{TR, TE, TM\}$$
$$TE = \{te_1, ..., te_n\}$$
$$TR = \{tr_1, ..., tr_n\}$$
$$TM = \{source, sink, global\_source\}$$

E consists of two sets, each containing edges of one specific type. D contains data communication edges, and C contains control edges. Each edge in E consists of pairs of tasks (u, v), representing a directed edge from u to v.

$$E = \langle D, C \rangle$$
$$D = \{(u, v, DATA) : u \in T \wedge v \in T \wedge u \neq main\_end \ \wedge v \neq main\_begin\}$$
$$C = \{(u, v, CONTROL) : u \in T \wedge v \in T \wedge u \neq main\_end \ \wedge v \neq main\_begin\}$$

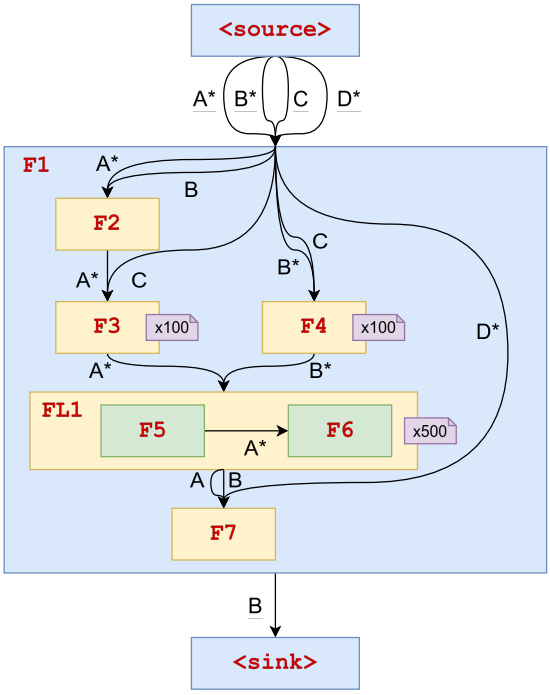
G is a multigraph where each task in TR may encapsulate tasks from $TR \cup TE$ and edges from E.

$$\Gamma : TR \rightarrow \mathscr{P}(TR \cup TE) \times \mathscr{P}(E)$$
$$\forall w \in TR : \Gamma(w) = (T_w, E_w)$$
$$T_w \subseteq (TR \cup TE) \setminus \{w\}$$
$$E_w = \{(u, v, l) : u \in T_w \wedge v \in T_w \ \wedge l = DATA \vee CONTROL\}$$
$$\forall w, x \in TR : T_w \cap T_x = \emptyset \wedge E_w \cap E_x = \emptyset$$

The ETG has two main types of tasks: regular and external. Regular tasks always map to a call site in the AST, and to the underlying function, which the subset guarantees is unique. External tasks, on the other hand, represent call sites to functions whose implementation is not available (e.g., calls to `printf`). Edges represent dependencies between tasks, and may contain information about a data item (i.e., a data object communicated to the task), or control information (e.g., whether the task is activated based on the result of an if/else block, or whether a task should

```c
void F1(int *A, int *B, int *C, int *D){
    F2(A, B);
    for (int i = 0; i < 100; i++){
        F3(A, C[i]);
        F4(B, C[i]);
    }
    for (int i = 0; i < 500; i++){
        F5(A, B);
        F6(A);
    }
    F7(A, B, D);
}
```

(a) A C program that follows the specified C subset



(b) The program's matching ETG representation, with three levels of hierarchy

Figure 3.6: Example of a C program that follows the subset, alongside its ETG.

be executed after a preceding task, even when they share no data dependencies). As the ETG is a multigraph, there may be multiple edges from one task to another.

The ETG is also hierarchical, i.e., a single task may abstract an entire subgraph, with its own subtasks and edges. This situation arises from the depth of the program's function call tree, which may have arbitrary depth, as well as from control flow structures on Type B functions. In Figure 3.6 (b), we show the ETG for the program in Figure 3.6 (a). This ETG has three levels of hierarchy: the top level has the special tasks `<source>` and `<sink>`, denoting the origin and destination of data, as well as the root task `F1`. The root task encapsulates the second level of hierarchy, consisting of 5 tasks. These map to 4 call sites in `F1` (`F2`, `F3`, `F4`, and `F7`), as well as one of its loops, `FL1`. This example highlights the two distinct ways loops are mapped on an ETG: if the call sites inside a loop represent independent tasks, i.e., have no data or control dependencies and all their pointers do not alias, those tasks remain at the same level of hierarchy, and are each annotated with its number of iterations. On the other hand, if the tasks have dependencies, we lower the hierarchy one more level by outlining the loop body with the call sites into a new function. In the example, this happens with the second loop, as tasks `F5` and `F6` have a data dependency. Task `FL1` represents the newly-created call site for the loop body function, and is annotated with the number of iterations.

3.3.1 Task Attributes

Our reference implementation of the ETG is built as a Clava extension. While the ETG is a generic representation that can be applied to contexts outside HW/SW partitioning, we specialize our implementation to include several task attributes useful for this scenario:

- **Percentage of computation time on CPU (%comp)**: the percentage of CPU time spent by a task, on a CPU, as measured through profiling using *gprof*. Due to the hierarchical nature of the ETG, the percentage of computation of a task should be close to the sum of the percentage of computation of its subtasks (discounting the imprecisions inherent with instrumentation-based CPU profiling);
  - **Software Execution Time (SWT)**: a derived attribute representing the CPU time of the task, as obtained by combining the percentage of computation with the total execution time of the application on the CPU.
- **Worst- and Average-Case FPGA latencies**: two estimates of latency obtained by synthesizing the task using Vitis HLS. Applications with dynamic data sizes require the annotation of each task's underlying function implementation with the maximum iterations of each loop. This is also obtained through CPU profiling, using instrumentation generated automatically by Clava;
  - **Hardware Execution Time (HWT)**: a derived attribute representing the FPGA time of the task, as obtained by combining the worst-case latency with the clock frequency obtained post P&R.
- **Hardware resources (%FF, %LUT, %BRAM, %DSP)**: the hardware resources required to implement the task on an FPGA, given as a percentage of total resources available on the target FPGA. These resources are flip-flops (FFs), Lookup Tables (LUTs), Block Random Access Memories (BRAMs), and Digital Signal Processors (DSPs). We further distinguish between resource usage post HLS and post P&R;
- **Synthesizability**: a list of non-synthesizable C-language features, as provided by Vitis HLS.

3.3.2 Edge Attributes

Each data edge in the task graph is annotated with the data item associated with it. Data items represent the arguments of the call site associated with a task, or the access to global variables. Data items may represent scalars, pointers and fixed-size arrays, as well as their identifying symbols (i.e., a call argument can have a different name than the function parameter at the same position). The size of a data item is relevant to the type of analysis performed over an ETG, but it is not always possible to know the size of a pointer's memory region at compile-time. By using profiling information, we can annotate all dynamic memory allocations on the AST with the maximum,

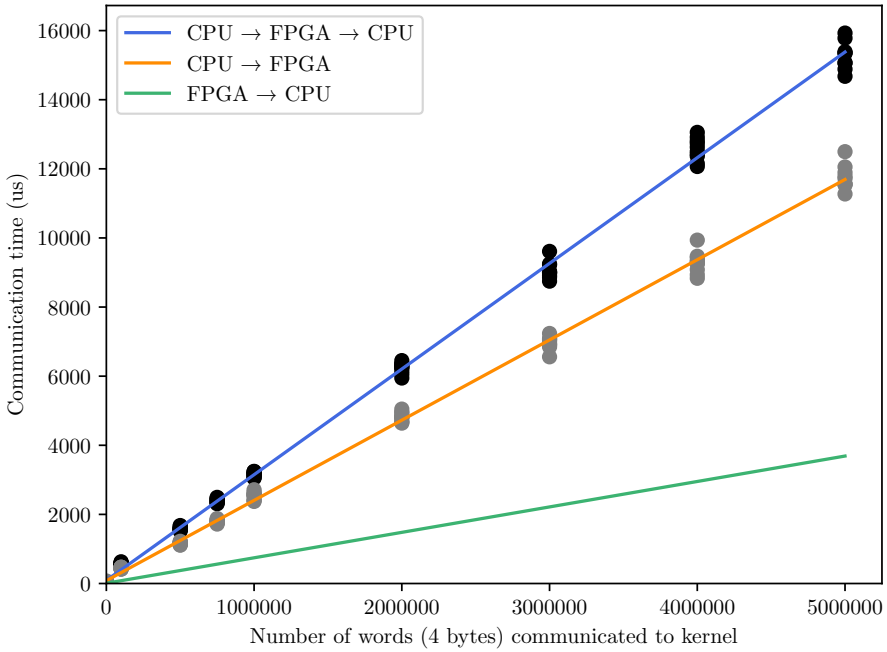


Figure 3.7: Example of a memory communication latency model for unidirectional $CPU \rightarrow FPGA$ and bidirectional $CPU \rightarrow FPGA \rightarrow CPU$ communication. Each input size is sampled 10 times for both scenarios using an Alveo 250. Models obtained through two linear regressions with adjusted $R^2 = 0.99$. The $FPGA \rightarrow CPU$ model is obtained by subtracting the previous two.

minimum and average size of their allocations for a sufficiently large input, and update the ETG with that information.

Knowing the size of each variable allows us to also estimate the communication cost between the source and target tasks. This communication cost exists when one of the tasks is allocated to hardware, and the other to software. We may estimate this cost by plugging the total size of each variable into a linear model that tells us, for any positive amount of bytes, the communication time. Figure 3.7 shows one such model, tailored to an HPC platform. This represents the cost of copying data from system memory to on-board memory, through PCIe transfers.

This model can be created for any CPU-FPGA heterogeneous system by repeatedly calling a kernel with ever-increasing input sizes, while measuring the time it takes for it to execute. This gives us the unidirectional $CPU \rightarrow FPGA$ communication time. We may also include an additional operation of copying the data back from the FPGA to the host, to represent the bidirectional $CPU \rightarrow FPGA \rightarrow CPU$ communication time. One necessary condition is that this kernel performs a fixed number of operations, independent of the input size. This allows us to subtract the constant kernel execution time from all measurements, leaving us only with the time spent in communication.

We measure, for many different input sizes, the execution time for both the unidirectional and bidirectional scenarios. Each input size is also sampled multiple times. We plug these values into a linear regression algorithm, which gives us a linear model for both types of communication. In

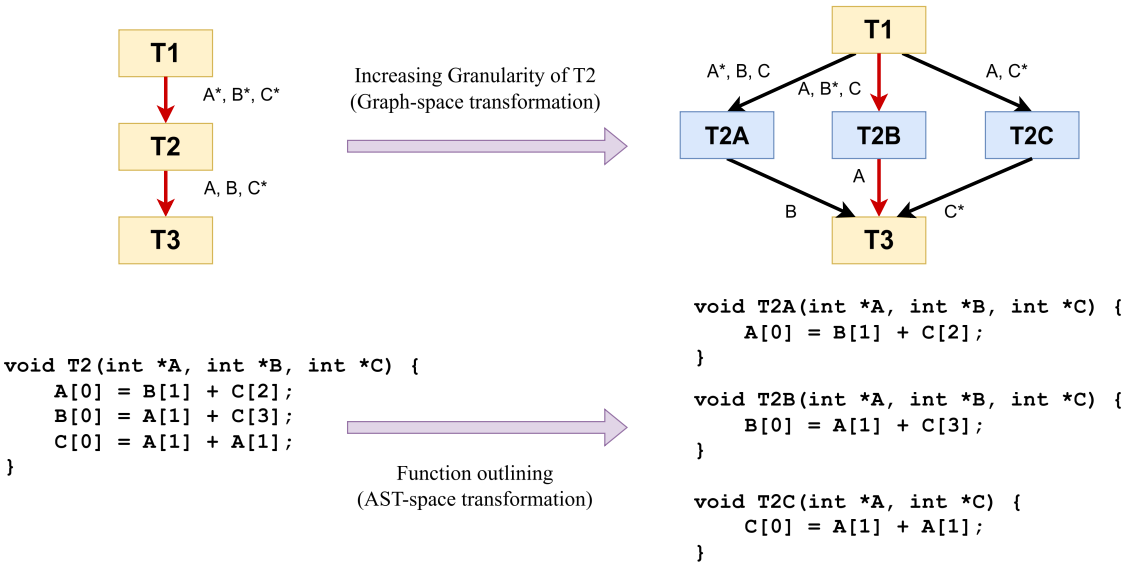


Figure 3.8: Example of how an increase in ETG granularity maps as function outlining transformations on the AST.

the context of an ETG, the bidirectional model is not outright useful, as edges always represent unidirectional communication. However, we may use it to obtain the cost of the $FPGA \rightarrow CPU$ communication by subtracting the bidirectional cost from the unidirectional $CPU \rightarrow FPGA$ cost.

Figure 3.7 shows that this cost can be much smaller than the cost of communicating data from the CPU to the FPGA. We suspect that this is due to the initial allocation of space on the FPGA on-board memory during the initial transfer, which does not occur when data is transferred back to system memory. We reinforce that this specific model pertains only to an HPC system, and does not account for communication costs between the FPGA and the on-board memory. It is also not relevant for embedded CPU-FPGA systems where both the CPU and FPGA share system memory, removing the need for moving data back and forth.

Each communication edge has, therefore, two separate unidirectional costs: the $CPU \rightarrow FPGA$ communication time, and the $FPGA \rightarrow CPU$ communication time.

3.3.3 Graph-space Transformations

As the ETG is an overlay over the program's AST, preserving a 1-to-1 mapping between tasks and call sites and their functions, any graph transformations applied over the ETG are reflected as AST transformations. This enables us to change the granularity of the ETG at the local level, i.e., to a single task or a small group of tasks, rather than the entire ETG. Each ETG transformation has, therefore, a matching AST transformation:

- Splitting a task into multiple tasks (i.e., increasing the local granularity) is mapped as the outlining of a function's code into new, smaller functions;

Table 3.1: Syntactical features of the selected applications from Rosetta [143] and MachSuite [118]

Application	#Tasks	Avg. #Statement	#for,while,if	% Static Loops
MS-backprop	23	18.68	41, 0, 1	78.95%
MS-fft-transpose	21	43.63	12, 0, 0	26.32%
MS-kmp	4	13.00	2, 2, 3	50.00%
MS-sort-merge	5	17.75	5, 0, 2	50.00%
MS-sort-radix	15	10.15	11, 0, 2	53.85%
R-3d-rendering	23	12.75	8, 0, 16	20.00%
R-digit-recognition	12	9.42	7, 0, 3	50.00%
R-face-detection	26	14.70	9, 1, 9	30.43%
R-optical-flow	9	33.00	25, 0, 11	88.89%
R-spam-filter	6	8.33	5, 0, 0	66.67%

- Merging multiple tasks into one (i.e., decreasing the local granularity) is mapped as the inlining of functions into one new, large function;
- Creating a new task creates a new function and call site, and deleting a task deletes its matching call site and function implementation, if it exists.

In Figure 3.8, we exemplify an increase in the granularity of task `T2`. On the original ETG, this task sequence is completely sequential, exposing no task-level parallelism. By applying task splitting to `T2`, however, we end up with three separate parallel tasks, which can be exploited by an FPGA. On the AST, this transformation is reflected as two function outlining transformations, which create two additional functions from the code of `T2`.

3.3.4 Characterizing Applications Using ETGs

To show the versatility of our ETG representation, as well as its generation using Clava, we select two benchmark suites, Rosetta (*R*) [143] and MachSuite (*MS*) [118]. Both benchmark suites have been used to evaluate FPGA optimizations and partitioning schemes of source code through HLS, which aligns with the purpose of our ETG. Furthermore, all Rosetta and some MachSuite benchmarks have multiple functions and complex control flow with call graphs with more than one level of depth, which makes them ideal for the call site-oriented hierarchical mapping of our ETG. MachSuite benchmarks consisting of a single function are not considered, as they lead to ETGs with a single task.

Potential for Granularity Changes

We start by looking at the number of tasks of each benchmark and their associated syntactical features, as shown in Table 3.1: using the proposed methodology, we find that the ETG with the largest number of tasks is obtained from *R-face-detection*, with 26 tasks. In the context of a binary partitioning problem, where each task can be independently assigned to HW or SW,

Table 3.2: Data Items and respective paths

Application	#Data Item	Avg. Size (bytes)	Avg. Read-Write Path Length	Avg. #Read-only Tasks/Path
MS-backprop	32	3346	2.66	0.53
MS-fft-transpose	32	667	2.25	1.13
MS-kmp	12	4055	1.50	0.42
MS-sort-merge	15	1096	1.27	0.87
MS-sort-radix	14	1795	2.00	1.21
R-3d-rendering	35	600	1.29	0.71
R-digit-recognition	23	28527	1.52	0.57
R-face-detection	108	2014	1.36	0.71
R-optical-flow	42	2326	1.00	1.05
R-spam-filter	20	923322	1.35	0.50

this theoretically implies 2^{26} possible solutions. While this is a manageable search space for some HW/SW partitioning formulations, it also allows us to explore the granularity of the ETG in several ways: not only could we increase/decrease the search space by increasing/decreasing the overall granularity of the ETG, but we could also preserve the search space's size by increasing the granularity of some tasks and decreasing the granularity of others, enabling the generation of better code regions for offloading without sacrificing the complexity of the partitioning problem.

Furthermore, we note some statistical information about the functions associated with each task. Firstly, the average number of statements per task (*Avg #Statements*) varies from 8.33 to 43.63. This may suggest that task graphs with more statements per task may be good candidates for an increase in granularity, and vice versa. Secondly, every benchmark has loops whose number of iterations cannot be determined statically, as shown by the percentage of statically bound loops *%SLoops*. This is due entirely to the limitations of static analysis, as all inputs have a fixed known size for all these benchmarks. This motivates the need for CPU profiling, as determining a maximum number of iterations for every loop is essential for HLS-centered analysis.

Data Paths

Table 3.2 shows the number of unique data items in each ETG, i.e., data created in one task and communicated to other tasks (data used only in a single task is ignored), as well as their average size in bytes. When compared to Table 3.1, we can start identifying some interesting scenarios, e.g., *R-digit-recognition* has an average number of tasks (12), each with very few statements on average each (9.42), but also has data items with a very large average size (28527). This may indicate that increasing the granularity of this ETG could lead to these large data items being communicated more often, which may be disadvantageous in a partitioning scenario.

Furthermore, we also consider two important metrics regarding the data path of a data item: the number of tasks in the Read-Write path, and the number of Read-only tasks, as shown in Figure 3.9. Let us assume the scenario where a data item `A`, created in `T1`, is communicated to tasks `T2` and `T3`. *T2*, in turn, writes to *A*, while *T3* simply reads from it. Now, let us assume another task, `T5`, requires *A* for any purpose, be it reading, writing, or both: in this case, we would be faced with a Read-after-Write (RAW) or Write-after-Write (WAW) dependency from *T2* to *T5*,

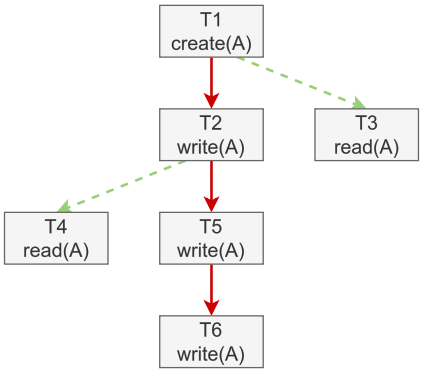


Figure 3.9: Example of the data path tree for data item `A`. Red edges represent the Read-Write path, and green edges represent the Read-only spurs.

which would be enforced by a communication edge. This can then be generalized to the whole ETG, which, due to its acyclic definition, ensures that there is a single sequence of tasks that have writes to a data item, from the moment it is created to the moment it is destroyed. We call this the Read-Write path, shown in solid red edges in Figure 3.9. In Table 3.2, we show the average length of the Read-Write paths for each benchmark, and ascertain that, on average, it is under 3 tasks. This has a paramount importance in partitioning, as clustering tasks around a Read-Write path eliminates the communication cost between its tasks.

As for the tasks that only have Read-after-Write (RAW) dependencies, such as `T3` and `T4`, we do not need to include any out-edges for data item `A`. In Figure 3.9, they are represented by the dashed green lines. Therefore, the datapath of a data item can be thought of as a tree, where its diameter consists of the tasks from the Read-Write path and whose leaves are the Read-only tasks. In Table 3.2, we can see that the average number of Read-only tasks on a data path is for the most part lower than its diameter, except for the ETG of *R-optical-flow*. This has partitioning implications, as Read-only tasks can potentially execute in parallel.

Functional Parallelism and Dataflow Regions

For each subgraph in an ETG, the compiler can determine the critical path of its tasks, in terms of data communication. Dividing the total number of tasks in the subgraph by the size of its critical path gives a relative measure *PL* of the subgraph's parallelism (i.e., a subgraph with $PL = 1$ would be made up entirely of a single sequence of tasks, e.g., $T1 \rightarrow T2 \rightarrow T3 \rightarrow T4$; and conversely, the higher *PL* is, the fewer dependencies there are between tasks, which increases the potential for executing them in parallel).

Table 3.3 shows the number of subgraphs in each ETG, the average size of their critical paths, and their average level of parallelism. We notice two extremes: *R-spam-filter* is completely sequential, with an average parallelism level of 1, while *R-optical-flow* is completely parallel. This provides valuable information: by increasing the granularity of a highly-sequential subgraph, we could expose some tasks that can be executed in parallel, and we could exploit highly parallel subgraphs by assigning their tasks to different devices during partitioning.

Table 3.3: Average parallelism level of each benchmark, and their task pairs that follow a producer-consumer relationship

Application	#Subgraph	Avg. Critical Path Length	Avg. Level of Parallelism	#Pairs	%Total Communication
MS-backprop	4	3.50	1.22	29	22.48%
MS-fft-transpose	5	3.80	1.03	9	7.76%
MS-kmp	1	2.00	1.50	2	11.11%
MS-sort-merge	1	2.00	2.00	1	4.55%
MS-sort-radix	2	3.50	2.00	9	20.45%
R-3d-rendering	5	2.20	2.02	6	6.32%
R-digit-recognition	3	2.33	1.83	3	6.67%
R-face-detection	7	1.71	2.07	7	3.27%
R-optical-flow	1	1.00	8.00	0	0.00%
R-spam-filter	1	4.00	1.00	3	10.34%

Producer-Consumer Relationships

Finally, we focus on finding pairs of tasks $Tx \rightarrow Ty$ where Tx communicates data to Ty following a producer-consumer relationship, i.e., Tx writes linearly to a data item, and from which Ty subsequently reads linearly. These patterns may indicate the presence of a dataflow region, which, depending on the underlying accelerator type, may allow us to execute both tasks simultaneously, as Tx produces data at the same rate and pattern as Ty consumes it. Table 3.3 shows how many pairs of tasks in the ETG follow this pattern, and how much that communication contributes to the total number of communication edges in the ETG. These values range from no patterns found in *R-optical-flow* to comprising 22.48% of the communication of *MS-backprop*, while also being present to some extent in the other ETGs.

3.4 Holistic HW/SW Partitioning

Once we have a task graph for an application, we can, at last, apply our holistic partitioning and optimization process. In the previous subsections, we have shown how individual tasks can be optimized, and how we can have an influence over the cost of communication by moving around data. Now, we provide the general foundations through which the partitioning itself can be done.

3.4.1 A Greedy and Heuristic Approach to Clustering Tasks

We envision our partitioning algorithm as a heuristic-driven greedy algorithm (and, therefore, a non-exact method). This is necessary, as the underlying number of tasks we have may change during the execution of the algorithm, and exact methods operate better when working with a fixed-size search space. Furthermore, we need a scalable solution capable of dealing with hundreds, if not thousands, of tasks obtained from large C applications. The previous subsection showed, in Table 3.1, that even small applications from Rosetta [143] and MachSuite [118] have up to 26 tasks, which contrasts with a solution space with an exponential complexity of $O(2^n)$.

Instead, we propose using a heuristic method. More specifically, we use a greedy algorithm to group tasks into a single cluster, which is then offloaded onto the FPGA. Starting from a single task, the algorithm performs a single pass over the task graph, greedily expanding the cluster with increasingly more tasks. As there can be m possible starting tasks, the algorithm creates multiple candidate clusterings, from which one is chosen. This reduces the problem down to a linear $O(n \times m)$. The heuristic guiding the expansion of the algorithm, as well as determining whether a task should be accepted, is at the core of our holistic process. Our ETG representation exposes properties from the entire application (e.g., whether a data item is used before or after a task executes, whether a task is synthesizable, which code optimizations can be applied), and helps inform one or more heuristics for the algorithm.

Finally, it is also important to define the boundaries of the algorithm, i.e., the first task in the cluster, and the exit condition. Starting with the latter, we replicate the process used by an expert developer to improve the performance of an application, and decide that we want a cluster that covers a percentage of the application's CPU execution time, e.g., 80%. By profiling the application on the target CPU, we determine which tasks most closely match the target percentage. These tasks serve as the upper bound for the expansion process, that is, the algorithm stops expanding once it reaches these tasks. As the ETG is hierarchical, the clustering process is done bottom-up, from the tasks at the lowest hierarchical level (i.e., our starting points), up to the hierarchies of the upper-bound tasks. Chapter 4 describes this algorithm in more detail.

3.4.2 Holistic Optimizations on a Cluster

Given a growing cluster, we can apply many types of code transformations and optimizations. They may involve individual tasks, groups of tasks, or even the whole cluster, and are the following:

- **Task-specific HLS optimizations**: these optimizations are applied only to individual tasks. Some examples of optimizations are loop unrolling, loop pipelining, and loop flattening. Function specialization could also be considered an intra-task optimization, even though it can only be applied once we compare its inputs with those of other tasks that contain instances of this function;

- **Enabling task synthesizability**: one of the most limiting factors during the greedy algorithm's expansion is the impossibility of including tasks that cannot be synthesized. As we work with large C applications that go beyond simple HLS-friendly kernels, we must deal with non-synthesizable C-language features, e.g., system calls (represented by tasks of type "EXTERNAL"), dynamic memory allocations, and indirect pointers. If the limitation is solvable through code transformations, then that allows the algorithm to include those tasks in the cluster. In Figure 3.10, we see this scenario in (a), where a call to `printf` prevents the clustering of that task. By transforming it into a synthesizable version of that system call, as in (b), we can expand the cluster in that direction;

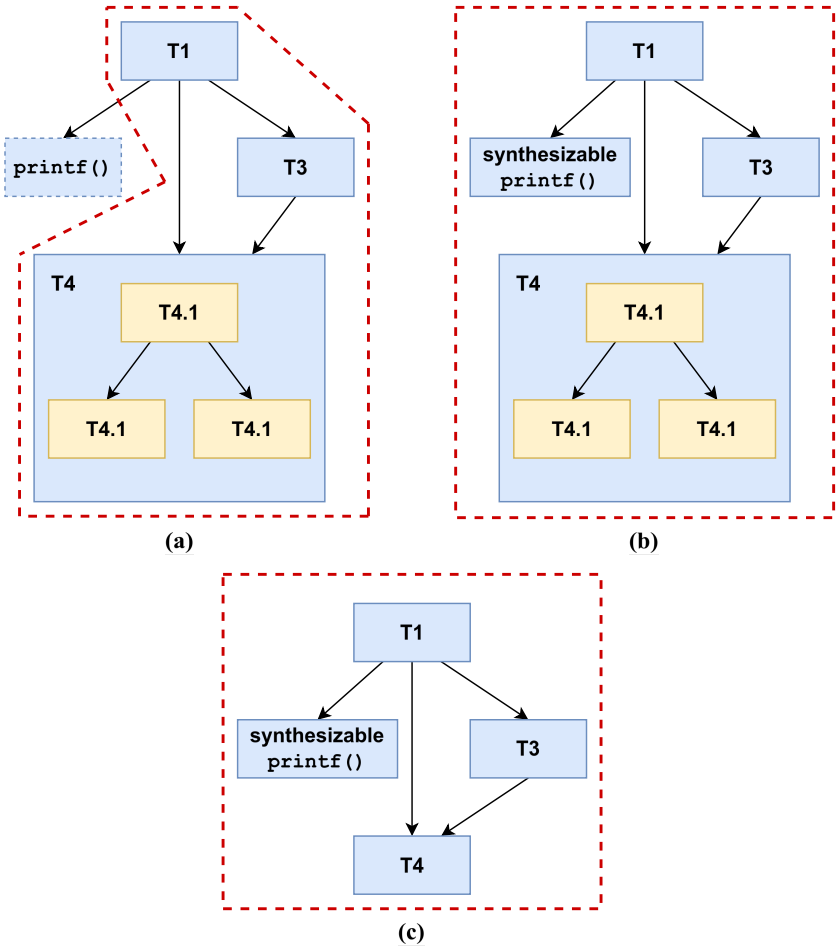
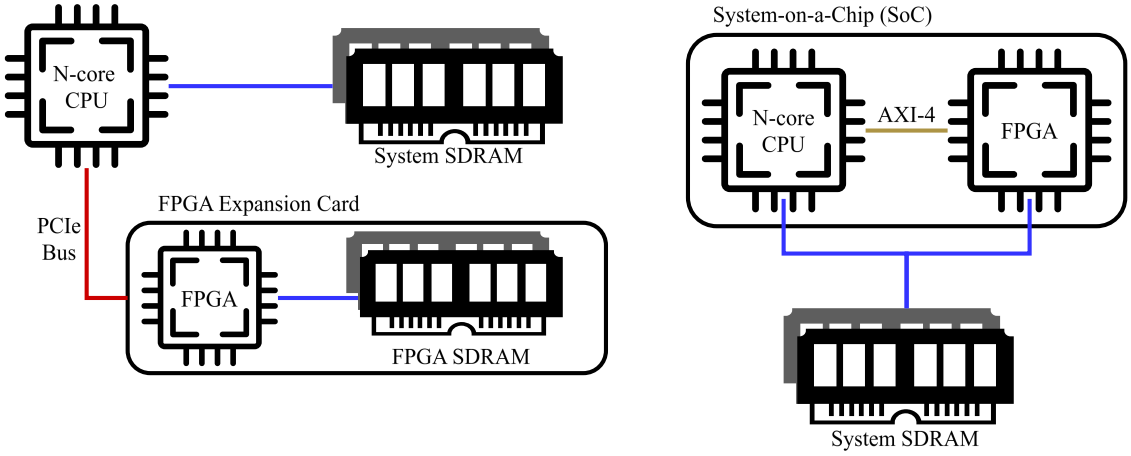


Figure 3.10: Example of how transformations enable clustering, firstly by transforming a non-synthesizable call to `printf` into a synthesizable one, and then by reducing the granularity of the ETG.

- **Reducing the granularity of the cluster**: as more and more tasks are added to the cluster, and the algorithm ascends the ETG hierarchy, it may be desirable to reduce the granularity of the cluster by applying task fusion. We see this in scenario (c) of Figure 3.10, where the inner tasks of `T4` undergo task fusion completely and remove the lowest level of hierarchy for this ETG. Recall that task fusion maps to function call inlining in the underlying AST;

- **Cluster-to-host optimizations**: these optimizations attempt to reduce CPU-FPGA communication and the overall memory footprint of the cluster. The major gain in this area is in adequately mapping data between the host and the FPGA cluster by minimizing memory accesses and maximizing the usage of on-chip memory. This relies heavily on a holistic analysis of both the FPGA cluster and the host tasks regarding buffer sizes, initializations, and whether any live-in and live-out buffers to and from the cluster can be discarded.

To conclude, let us reinforce the holistic component of this approach: by definition, a holistic approach is one that attempts to explain the interplay of individual components in a system by



a) High-power Computing Heterogeneous CPU/FPGA System

b) Embedded Heterogeneous CPU/FPGA System

Figure 3.11: Architecture of a heterogeneous CPU and FPGA system, in its HPC (a) and embedded (b) configurations. Blue lines represent memory accesses, red lines a PCIe bus, and yellow lines represent an AXI-4 interface.

looking at the system as a whole. Recontextualizing this definition in the context of HW/SW partitioning, we have that the individual decisions made about the offloading of any task, and its related optimizations, can only be decided — and improved — by examining what their impact is on the total execution time of the application.

3.5 Evaluation Strategy

We now describe the steps taken to evaluate our algorithm. We start by defining the target platform over which we validate the results, followed by a description of the benchmarks used. Finally, we define the metrics measured for the purposes of evaluation.

3.5.1 Target Architecture

There are two main approaches to heterogeneous CPU-FPGA systems: High-performance Computing (HPC) systems and SoC-based embedded systems. Both types of systems contain one or more CPU cores, and one or more FPGA programmable logic (PL) units; the differences do not lie with the number, but rather with how powerful each of these components is, as well as how much memory is available.

In heterogeneous HPC CPU-FPGA systems, the programmable logic is often provided in the form of PCIe expansion cards. These cards can hold powerful FPGAs capable of running at higher frequencies, as they have dedicated cooling independent of that from the CPU. These cards can also come with SDRAM of their own, which is independent of system memory. This means that the FPGA can work with more data than what fits into its BRAMs, without incurring the performance penalty of having to access system memory. However, this varies from system to system, as it is also possible to efficiently share system memory between a CPU and a PCIe FPGA

accelerator [151]. Furthermore, the CPUs in these systems also tend to be powerful (i.e., server-grade), and capable of running at very high frequencies as well. The drawbacks of these systems come from their large size and high energy demands. These kinds of systems can usually be found in data centers, servers, and workstations, which in turn makes them popular for cloud computing.

Table 3.4: Comparison of two HPC and embedded platforms, in terms of the CPU and FPGA properties and the different types of memory available to them

		HPC System Alveo U250 + AMD CPU	**Embedded System Zynq MPSoC ZCU102**
CPU	Model	AMD Threadripper 3960X	ARM Cortex-A53 + Cortex-R5F
	ISA	x86_64	ARMv8-A
	Core Count	24	6 (4 + 2)
	Frequency (MHz)	2193	600
FPGA	Model	XCU250	XCZU9EG-2FFVB1156
	#Flip-Flops	2749000	548160
	#Lookup Tables	1341000	274080
	#Block RAM	2000 32K BRAMs	1824 18K BRAMs
	#DSP	11508	2520
System	FPGA SDRAM (GB)	64	—
	System SDRAM (GB)	32	4
	CPU-FPGA Interface	PCIe Gen3 x16	AXI4
	Operating System	Ubuntu 22.04	PetaLinux 2024.2

On the other hand, embedded systems are often built as systems-on-a-chip (SoCs). These chips combine the CPU cores and the FPGA into one die, which allows for a single cooling solution in a small-sized board. These SoCs, while requiring less energy than their HPC counterparts, also run at lower clock speeds. Furthermore, all system memory is shared between the CPU and the FPGA, as there is typically no memory dedicated to the FPGA besides its BRAMs. Modern SoCs may include UltraRAMs (URAMs), which are reconfigurable blocks with higher latency than regular BRAMs, but lower latency compared to an off-chip memory access. Their low energy usage, as well as their small form factor, makes them suitable for deployment in a wide variety of systems, making them popular as components of cyber-physical systems and edge computing networks. Figure 3.11 summarizes the architectures of both types of systems. Table 3.4 describes two systems, one HPC and the other embedded, contrasting their available resources and computational power.

In some HPC systems, including the one described, the CPU's single-thread performance may be high to the point of limiting what can be extracted by the FPGA. HPC-based FPGA devices take the most advantage out of high-throughput applications running indefinitely, and are difficult to exploit for finite regions. Embedded systems, on the other hand, have much better parity between the CPU and the FPGA. Taking the ZCU102 platform described in Table 3.4 as an example, its ARM CPU is both simpler and underclocked when compared to an HPC counterpart, bringing it closer to the frequency of an FPGA, which operates in the 150 MHz to 250 MHz range. For these reasons, we believe our approach benefits the most by targeting the embedded space, and choose the ZCU102 as our target platform.

3.5.2 Benchmarks

Now, we briefly describe the benchmarks used for evaluation:

- Three handpicked benchmark applications:
  - *edgedetect*: part of UTDSP [152], this application, while simple, serves to illustrate some examples throughout this thesis;
  - *spam-filter*: part of Rosetta [143], previously characterized in Table 3.1. We use this synthesis-friendly application as an example of how we can use synthesizable calls to `printf` for HLS debugging;
  - *llama2*: an implementation [139] of the inference part of Llama2, an LLM [153]. We use it as an example of a large application that performs calls to `printf`, motivating the need for a synthesizable alternative.
- All nine computer vision applications from the SD-VBS subset [154] of the CortexVision [137] benchmark suite:
  - *disparity*: computes depth information from a pair of stereo images by calculating pixel displacement;
  - *localization*: estimates a mobile robot's position and orientation using a Monte Carlo particle filter;
  - *mser*: identifies stable blob regions across varying thresholds for robust region matching;
  - *multi-ncut*: partitions an image into distinct perceptual regions using graph-based segmentation;
  - *sift*: detects and describes scale- and rotation-invariant local features for image matching;
  - *stitch*: blends overlapping images together to construct a single seamless panorama;
  - *svm*: classifies feature vectors using a trained decision boundary;
  - *texture-synthesis*: generates a larger, non-repeating texture image given a small input sample;
  - *tracking*: tracks point features across sequential video frames using the Kanade-Lucas-Tomasi algorithm.

3.5.3 Evaluation Metrics

The final evaluation of our algorithm uses the CortexSuite applications, as well as *edgedetect*. We profile the applications on the ZCU102 board's CPU, obtaining the percentage of CPU computation time spent on each call site. This allows us to pinpoint a root task to generate an ETG from, as shown in Table 3.5. For all benchmarks, the top-level task is a call site in `main()`. While some of

Table 3.5: Profiling of the selected applications on the ZCU102 board, with identification of each program's useful region

Application	Input Size	CPU Exec. Time (s)	ETG Root Task Call Site	%Comp
disparity	fullhd	19.60	`getDisparity`	99.2
localization	vga	1.47	`updateState`	99.7
mser	fullhd	3.63	`mser`	94.0
multi-ncut	qcif	51.09	`segment_image`	100.0
sift	fullhd	18.67	`normalizeAndSIFT`	98.1
stitch	fullhd	175.03	`stitch`	100.0
svm	cif	0.98	`svm`	98.9
texture-synthesis	fullhd	110.11	`create_texture`	98.7
tracking	fullhd	4.56	`trackFeaturesPyramidalLK`	78.6
edgedetect	fullhd	0.31	`edge_detect`	92.0

%Comp: percentage of computation (i.e., CPU execution time) covered by the root task.

these call sites, such as the ones in *multi-ncut* and *stitch*, represent 100% of the computation time, other applications have call sites from 92% to 99.7%. This is due to some computations performed in `main()`, such as file I/O and data integrity checks, that are of no interest to the ETG, and the underlying partitioning problem.

As all applications rely on dynamic data, being suitable for images of different standard resolutions, we settle on the largest possible size for all applications. If we support the largest input, then the smaller inputs are also supported by definition, as the only things that change are loop bounds and number of iterations. Table 3.5 shows the execution time, as an average of 10 runs, for each application using the largest input sizes. The only exception is in *multi-ncut*, which fails to finish in a practical timeframe when using the largest input available (i.e., ≈4 days per run); we choose the second-largest size instead.

The evaluation of our algorithm combines different clustering heuristics, code transformations, and optimizations. For each set, we measure and report on the following:

- The execution time speedup achieved by the offloaded region when compared to its equivalent in the CPU baseline, calculated using the worst- and average-case latencies and the FPGA frequency achieved post P&R;

- The execution time speedup of the entire CPU-FPGA application when compared to the total execution time of the CPU baseline;

- The resource usage post P&R, with a particular focus on BRAM usage across different optimizations;

- The code complexity metrics, obtained through Frama-C, for the offloaded region, comparing the metrics before and after applying transformations on the cluster.

3.6 Motivating Example

To better illustrate the advantages of a holistic partitioning and optimization process, we use the *edgedetect* application from our benchmark selection. We show its detailed call graph in Figure 3.12. This application takes a grayscale image with resolution $W \times H$ as its input, and it outputs a black-and-white image of the same resolution that highlights the edges detected in the original. Furthermore, we add an early step to convert an RGB color image into grayscale. To do this, the application performs the following steps:

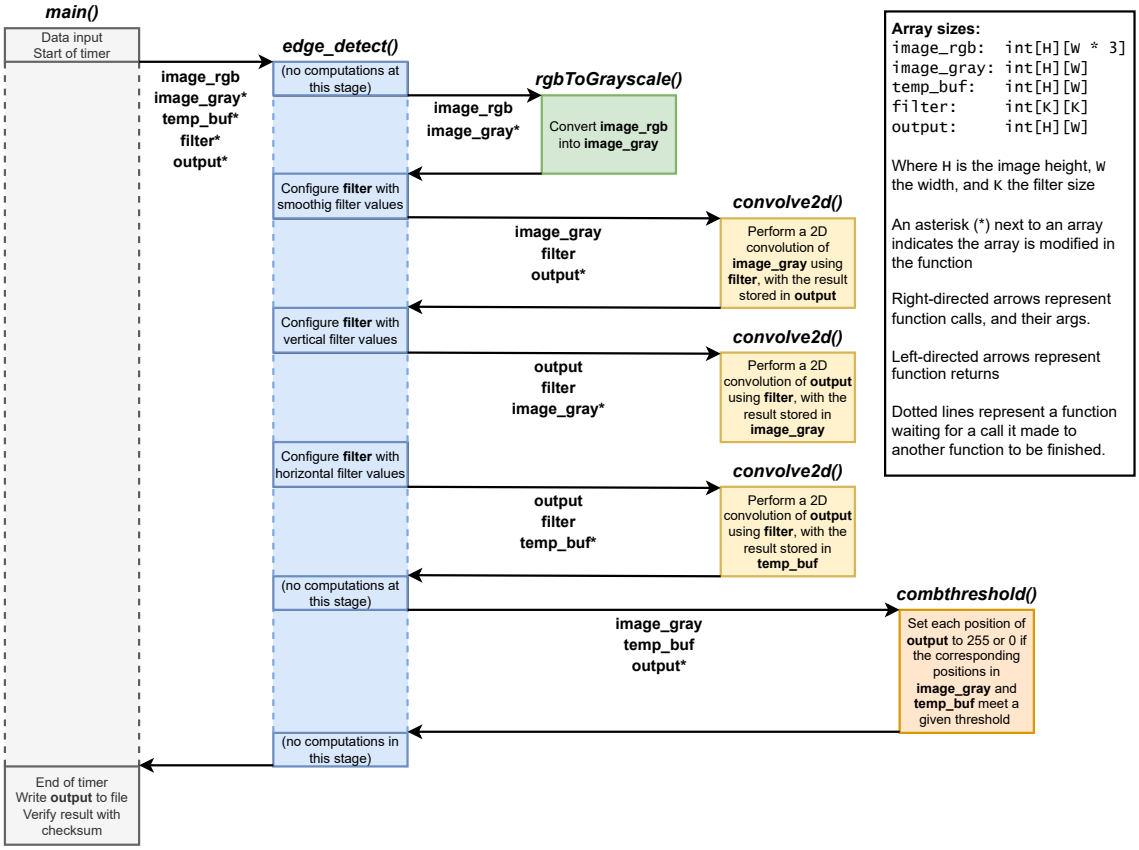


Figure 3.12: Function call graph for the *edgedetect* application. The code preprocessing steps have already been applied except for flattening the 2D arrays, which have been preserved for readability purposes.

1. It starts by providing the `edge_detect()` function with all the required data and buffers it needs: the input image `image_rgb`; three intermediary buffers, `image_gray`, `temp_buf`, and `filter`; and `output`, a buffer to store the final result;

2. It starts by converting the RGB image to a grayscale image, in function `rgbToGrayscale()`. It calculates the gray component as a weighted sum of the three color components, which means that `image_gray` has $1/3$ the size of `image_rgb`. The other buffers also have this size, as `image_rgb` is not used past this function;

3. In this step, we blur the grayscale image using a smoothing filter. We set `filter` with the values of a Gaussian smoothing filter, and call the `convolve2d()` function to apply a convolution using that filter. This filter is a square matrix of size $K \times K$;

4. Next, we set `filter` to have the values of a Gaussian filter capable of highlighting vertical edges, and call `convolve2d()` once again to apply the convolution with the new filter. The result is stored in `image_gray`;

5. Now we repeat the same step, but for a horizontal edge highlighting filter instead: we set `filter` to the correct values and call `convolve2d()` a third time. The result is stored in `temp_buf`, so as not to overwrite the buffer with the highlighted vertical edges;

6. Finally, we combine the results in `image_gray` and `temp_buf` using `combthreshold()`. In this function, if a given pixel in `image_gray` or `temp_buf` has a value above a certain threshold (meaning that the pixel is part of an edge), the corresponding pixel in `output` is set to white; otherwise, it stays black. Once this is done, `output` will have the final result of the application.

We further specialize this application to use images with a resolution of 512×512 pixels, and filters of size 3×3. Using static sizes is advantageous: on one hand, it makes it trivial to determine the number of bytes of each array, which allows us to determine how much time is required to communicate them; and on the other, it leads to more and improved HLS optimizations, as we can now apply array partitioning over each array, while also enabling more informed choices for loop unrolling. We name this baseline version of the *edgedetect* application **ED-B**.

Despite these advantages, not every application relies exclusively on static data sizes. In the upcoming subsections, we touch briefly on how we could handle that scenario.

3.6.1 Characterizing the Task Graph

If we generate a task graph using the function call graph as a basis, we end up with the graph in Figure 3.13. We note that the `edge_detect()` function from the task graph was turned into three separate tasks (`set_smooth_filter`, `set_vert_filter` and `set_horiz_filter`), as per the graph generation process we previously described. We also note that the three calls to the `convolve2d` function have also been turned into three identical tasks, with a number identifying each specific instance. These duplicated tasks all share the same attributes (execution times and hardware resources), but have different inputs.

We generate the metrics for each task using the methodology we previously mentioned: CPU execution times come from profiling, and FPGA execution times, as well as the predicted resource usage of the hardware design, come from the HLS report. We calculate these values using the previously described embedded system as our target, and choose an FPGA frequency of 100 MHz. We apply some of the HLS source code transformations specified in [145], in order to obtain plausible hardware models (synthesis with no optimizations often leads to models with no useful

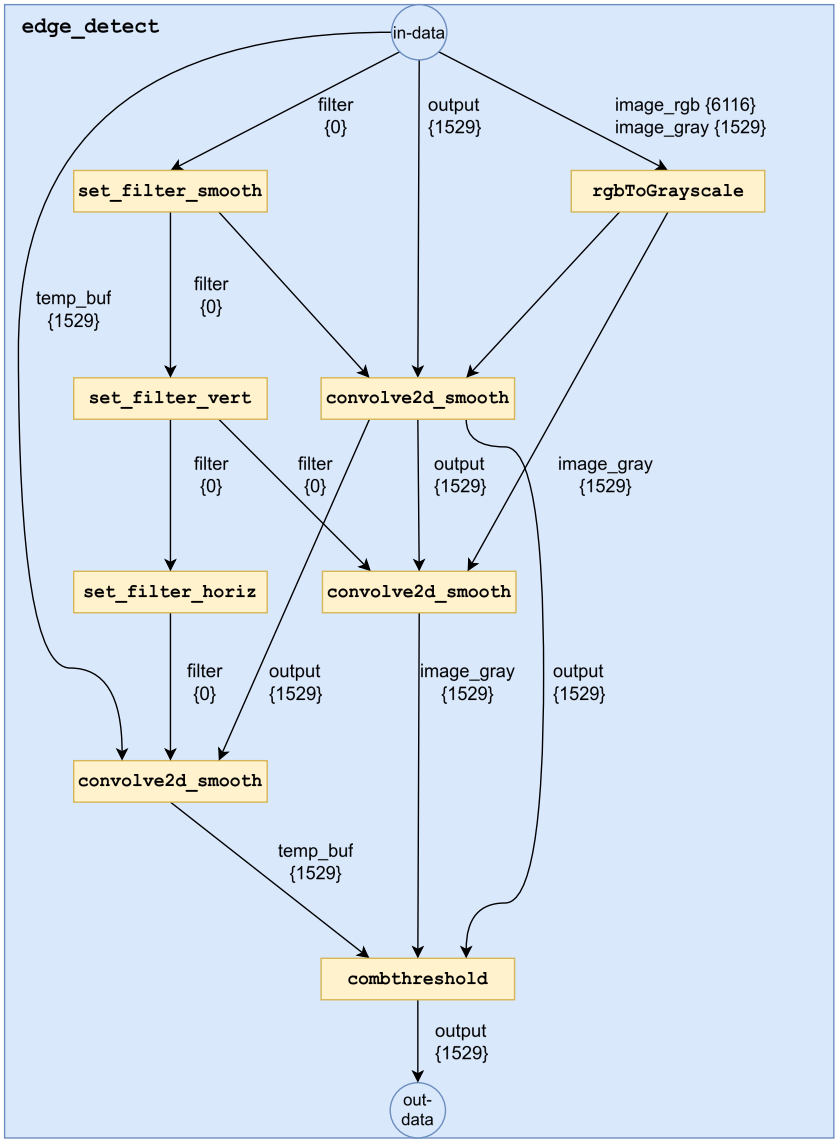


Figure 3.13: Task Graph for the ED-B version of the *edgedetect* application, with a fixed image size of 512×512 pixels. Edges represent the communication cost of the associated data.

performance). These consist, in the context of this application, of completely unrolling the innermost loops with a small number of iterations (i.e., the loops with K iterations), while pipelining the loops that work on the $H \times W$ image.

We show the ETG task attributes in Table 3.6. We note that all the `convolve2d` tasks all share the same implementation: this means that the hardware resource cost of having one of them in hardware is the same as having multiple, as we can reuse the same hardware model for each instance.

Each edge of the task graph is also annotated with the time it takes to communicate its associated data. We are assuming the worst case scenario where we choose the highest value out of the two communication directions, i.e., $CPU \rightarrow FPGA$ and $FPGA \rightarrow CPU$. Buffer `filter` has a communication cost of 0, which is due to it having a size of only 36 bytes (9 elements $\times$ 4-byte

Table 3.6: Metrics for the tasks in the ETGs of ED-B, ED-O1, and ED-O2

Version	Task	SWT (us)	HWT (us)	%FF	%LUT	%BRAM	%DSP
ED-B	rgbToGrayscale	143601	7866	1.33%	5.05%	0.00%	0.56%
	convolve2d (1)	14467	23414	2.65%	13.34%	0.00%	1.07%
	convolve2d (2)	14467	23414	2.65%	13.34%	0.00%	1.07%
	convolve2d (3)	14467	23414	2.65%	13.34%	0.00%	1.07%
	combthreshold	245034	5244	7.83%	2.16%	0.00%	0.00%
	set_filter_smooth	0	1	0.34%	1.04%	0.00%	0.00%
	set_filter_vert	0	1	0.34%	1.04%	0.00%	0.00%
	set_filter_horiz	0	1	0.34%	1.04%	0.00%	0.00%
ED-O1	rgbToGrayscale	149827	7866	1.33%	5.05%	0.00%	0.56%
	convolve2d_smooth	17706	23410	0.34%	1.04%	0.00%	0.00%
	convolve2d_vert	17706	15607	1.36%	1.63%	0.00%	0.00%
	convolve2d_horiz	17706	15607	0.30%	0.99%	0.00%	0.00%
	combthreshold	270074	5244	7.83%	2.16%	0.00%	0.00%
ED-O2	rgbToGrayscale	149827	7866	1.33%	5.05%	0.00%	0.56%
	convolve2d_smooth	17706	23410	0.34%	1.04%	0.00%	0.00%
	convolve2d_fused	22430	20809	0.41%	1.01%	0.00%	0.00%
	combthreshold	270074	5244	7.83%	2.16%	0.00%	0.00%

words). The linear communication model cannot accurately predict the cost of communicating values this small, as it itself is trained on timing measurements that do not show any measurable difference between communicating nothing and communicating only a few bytes.

The results we obtain using this motivational example let us reflect, then, on the necessity - and pitfalls - of adopting a holistic approach to combining HW/SW partitioning with code optimizations: every optimization we apply during partitioning needs to be made with both a global and local view of the application to be successful, showing that there is indeed a lot to gain by combining both approaches.

3.6.2 Application Optimization

Now that we have introduced how the *edgedetect* application works, we can start considering some opportunities for optimization. These opportunities consist of things that would, ideally, be detected by our holistic partitioning and optimization process, and could not be as thoroughly exploited by independent processes that look at each task individually. We note the following two opportunities:

- Taking a look at the task graph in Figure 3.13, we notice that all the required buffers come from the `main_begin` task, which we have previously defined as being a special task that is always assigned to software. This means that we will incur a communication cost between `main_begin` and the first task that uses each of the buffers. This is further compounded by all of these buffers being empty, except for `image_rgb`. A more optimal implementation would, then, only declare these buffers when they are needed. If we apply this to our case, the only buffers we strictly need to communicate from the software `main_begin` task are,

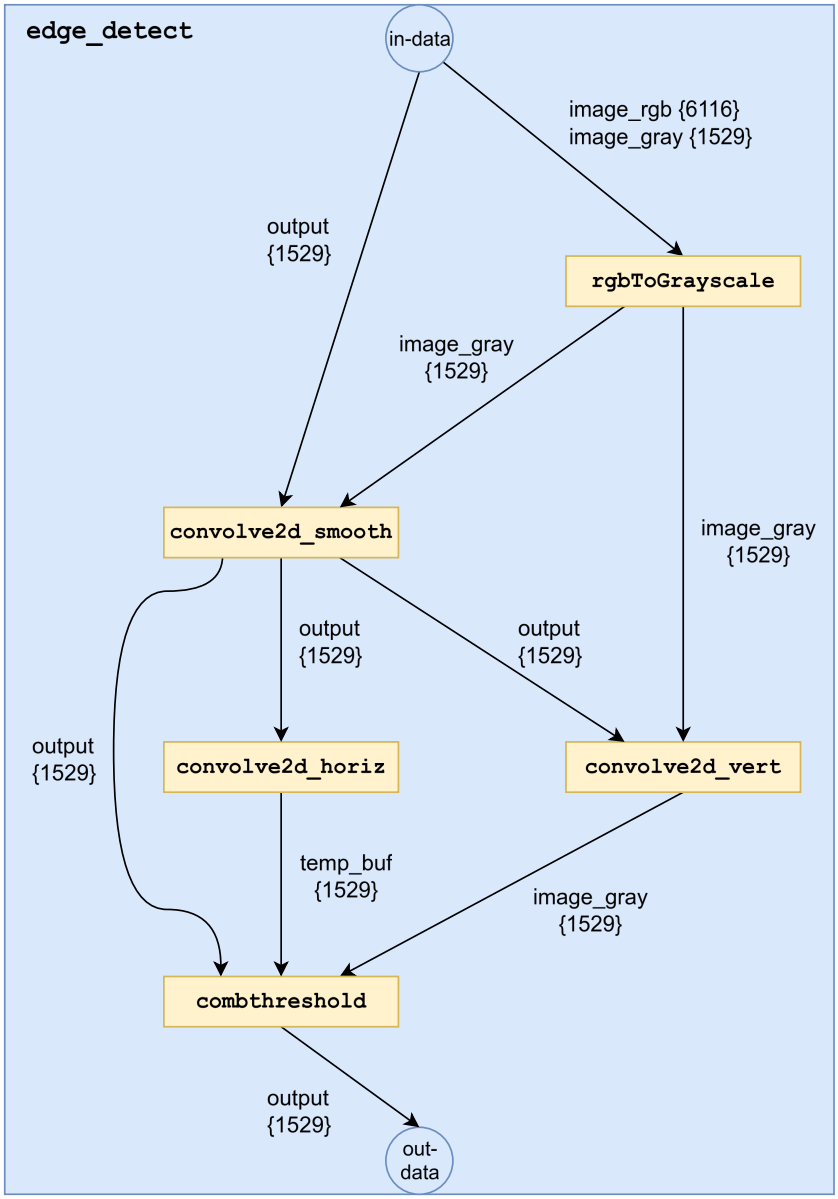


Figure 3.14: Task Graph for the ED-O1 version of the *edgedetect* application. Data obtained for a fixed image size of 512 × 512 pixels, with their communication cost represented on the edges.

then, the input image and the output buffer. All other buffers - `image_rgb`, `temp_buf`, and `filter` - are created only when they are needed;

- All of the `convolve2d` tasks require the buffer `filter`, which is filled with different values in each instance. Instead of having to set these using the `set_X_filter` tasks, we could, instead, have specialized versions of `convolve2d`, with hard-coded filter values. We call these `convolve2d_smooth`, `convolve2d_vert`, and `convolve2d_horiz`. If we do this, we not only remove all of the `set_X_filter` tasks, but also open the possibility of applying different optimizations to each specialized task. For instance, if one of the hard-coded filters has one or more zeros, we can skip some multiplications in the convolution.

This leads us to the task graph in Figure 3.14, and whose equivalent function call graph is depicted in Figure C.1 of Appendix C. The attributes of its tasks are depicted in Table 3.6, which are measured in the same embedded platform as the baseline version. We name this optimized version of the *edgedetect* application **ED-O1**. It represents, then, a version of *edgedetect* that minimizes communication and specializes tasks when we assume nothing about which tasks will be offloaded to each device.

However, we may take this one step further and start assuming that certain tasks will be assigned to the same device. As an example, tasks `convolve2d_horiz` and `convolve2d_vert` have the same exact computations: they apply a 2D convolution, pixel by pixel, by reading the image present in `output`. The only differences are that they write the result to `temp_buf` and `image_gray`, respectively, and that they use different embedded `filter` values for the convolution.

Given this similarity, and assuming that we know these two tasks are assigned to the same device, we can then fuse the two tasks together. If we fuse the tasks, we end up with a single task, `convolve2d_fused`, that has a single loop that calculates two convolutions in each iteration. This loop is obtained by fusing the loops of the previous two tasks, and is therefore an optimization that can only be applied when knowledge about the partitioning is available.

To this new version of *edgedetect* we give the name of **ED-O2**. Figure 3.15 depicts its task graph, and whose attributes are shown in Table 3.6.

3.6.3 Application Partitioning

Before we apply partitioning, we need to gather some baseline results first. For all 3 versions of the *edgedetect* application, we measure what we call the "trivial partitionings". These consist of the result of running a pure software version of the application (P-SW), and the result of synthesizing the whole application to the target FPGA as a single kernel (P-HW). These results are gathered in Table 3.7. For the software version, there is very little difference in terms of speedup between ED-O1 and ED-O2 when compared to the baseline ED-B. However, the hardware version shows a speedup of 31× for ED-O1 and 42× for ED-O2 when compared to ED-B. This means that the memory and specialization optimizations we apply to the source code already lead to improvements in performance even before partitioning.

Now that we have some baseline results, we can apply partitioning to all three versions. To do this, we use an ILP-based formulation of the HW/SW partitioning problem, which we describe in Appendix C. This formulation takes into consideration all the attributes in each task of the task graph, and aims to minimize the total execution time of the application (i.e., the SWT or HWT cost for each task, depending on the solution being evaluated, plus the cost of communication if two tasks are in different devices). This formulation is used here for illustrative purposes, and in lieu of the algorithm proposed in this project. However, this ILP formulation, or an evolution of it, may prove itself useful in the long run, as it may serve as a way for us to extract baseline results (as previously described in the evaluation section).

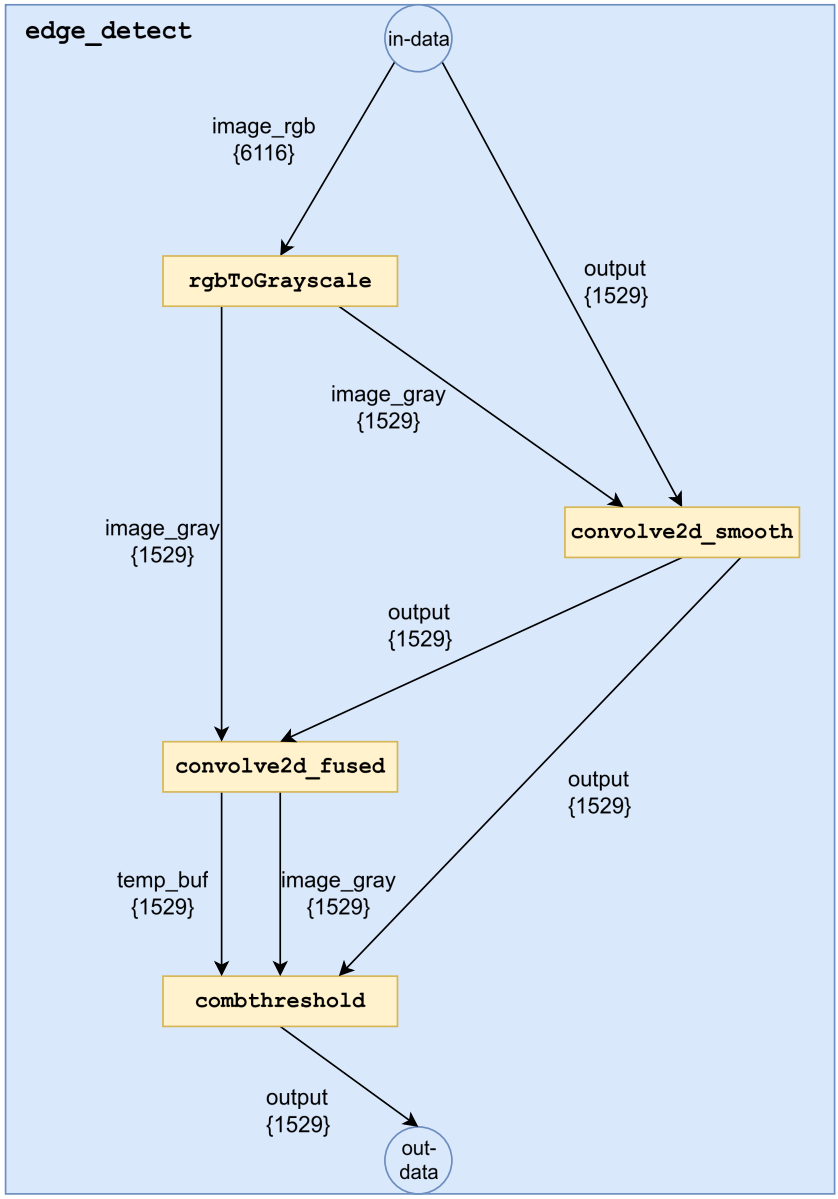


Figure 3.15: Task Graph for the ED-O2 version of the *edgedetect* application. Data obtained for a fixed image size of 512×512 pixels, with their communication cost represented on the edges.

The results of partitioning for ED-B are presented in Table 3.8. As we can see, the algorithm decided that it should offload the `rgbToGrayscale` and `combthreshold` tasks to hardware, while keeping the rest in software. This leads to a modest speedup of $6.1\times$ when compared to the pure software version, and a large speedup of $97\times$ when compared to the pure hardware version. This partitioning is, then, successful at providing a partitioning with better performance than the naive approach. We also point out that our ILP formulation allows us to extract how much time is spent in communication in relation to the total execution time of the application.

We may also notice the emergence of preliminary "clusters" of hardware and software tasks: the `convolve2d (N)` tasks and the `set_filter_X` tasks all work in tandem, alternating with

Table 3.7: Execution time of the baseline and optimized versions of the *edgedetect* application, when applying two trivial HW/SW partitioning schemes: everything to software (P-SW), and everything to hardware (P-HW)

Version	Trivial Partitioning	ExecTime (us)	Speedup vs. ED-B
ED-B	P-SW	435,372	–
	P-HW	1,464,479	–
ED-O1	P-SW	475,604	0.92×
	P-HW	46,924	31.21×
ED-O2	P-SW	430,172	1.01×
	P-HW	36,520	40.10×

each other. This causes a high degree of communication between them, which means that they benefit from being in the same device. This cluster is then bookended by the two hardware tasks, `rgbToGrayscale` and `combthreshold`. On a topologically sorted task graph, these would be the first to be executed after `main_begin` and the last to be executed before `main_end`, respectively.

Moving on to ED-O1, we can find the results of its partitioning in Table 3.9. In this case, the ILP formulation decided that every task should go to hardware. This is particularly interesting: if we refer back to the task graph properties in Table 3.6, we can see that, in ED-O1, `convolve2d_vert` and `convolve2d_horiz` have a smaller HWT than their equivalents in ED-B (15,607 ns and 23,414 ns, respectively). This is due to fewer operations in the horizontal and vertical specializations of the convolution, as some of the filter values are 0. Finally, and perhaps most importantly, we note that despite there not being any sizable difference between the HWT of the generic `convolve2d` and its specialized equivalent `convolve2d_smooth`, they still ended up offloaded to different devices: to software on the former, and to hardware on the latter. This raises an important point: while we may not do anything that directly improves the individual performance of a task, when we look at it as a whole, together with its neighboring tasks, we may end up changing that task's expected target device. Conversely, introducing measurable optimizations in one task may lead to changes in the final device assigned to it and its unoptimized neighbors.

The decreased HWT of these tasks, coupled with no longer needing to communicate data to and from the `set_filter_X` tasks, which have disappeared, causes the ILP formulation to also offload the convolutions to hardware. In terms of comparisons to the ED-B baseline, ED-O1 outperforms both its trivial solutions and its ILP solution, achieving a speedup of 2.2×. This is due in part to the specialization of the tasks, but also by decreasing the communication, as ED-O1 only spends 7,570 us in communication, as opposed to the 15,090 us spent by ED-B (a decrease of 49.8%). Curiously, despite this reduction in communication, ED-O1 spends more of its total time in communication (23.44%) than ED-B (21.08%). This only reinforces the interplay between minimizing communication and optimizing tasks, as by improving one we can improve the other, and vice-versa.

Finally, we look at ED-O2's partitioning, whose results are in Table 3.10. Results are similar,

Table 3.8: Results of applying an ILP-based HW/SW partitioning (P-ILP) to ED-B

Task	Device
rgbToGrayscale	HW
convolve2d (1)	SW
convolve2d (2)	SW
convolve2d (3)	SW
combthreshold	HW
set_filter_smooth	SW
set_filter_vert	SW
set_filter_horiz	SW
main_begin	SW
main_end	SW
Total exec time (us)	71,601
Total comm. time (us)	15,090
% of comm. time	21.08%
Speedup vs. P-SW	6.1×
Speedup vs. P-HW	97.0×

but slightly worse, as the execution time increased when compared to ED-O1 (despite still being better than the baseline). This can be attributed to the increase in communication cost, which happens despite us having removed one entire task through task fusing. This raises yet another important observation about task fusing: while we may end up with a task whose execution time is less than the sum of the execution times of the two tasks that gave it origin, this may not be in the best interest of the global whole of the application, as we are paying back the execution time we saved by increasing the communication cost.

Impact of Dynamic Array Sizes

Finally, we also want to consider the case when we do not know the size of the input. We rewrite the *edgedetect* application so that it accepts input images whose size is not known at compile time (e.g., `image_gray[H * W]` becomes `image_gray*`). For illustrative purposes, we synthesize everything to hardware (i.e., we apply a P-HW trivial partitioning). We do this for ED-B and ED-O1, which we rename as ED-B-D and ED-O1-D, as they now use dynamic data.

Converting static data to dynamic creates two limitations: firstly, the number of loop transformations we can do is much reduced, as we do not know their number of iterations. Pipelining is not much affected, but it makes it so that we cannot get a latency estimation out of our models (recall that latency estimation is how our HW/SW partitioning scheme determines the hardware execution time of a task).

The second issue is that all buffers that do not get communicated as an input, such as `temp_buf`, need to be allocated with an adequate size. We have two options here:

- We allocate these buffers in `main_begin` and pass them as inputs to the hardware tasks, which leads to a higher communication cost (this is the behavior of the baseline *edgedetect*);

Table 3.9: Results of applying an ILP-based HW/SW partitioning (P-ILP) to ED-O1

Task	Device
rgbToGrayscale	HW
convolve2d_smooth	HW
convolve2d_vert	HW
convolve2d_horiz	HW
combthreshold	HW
main_begin	SW
main_end	SW
Total exec time (us)	32,294
Total comm. time (us)	7,570
% of comm. time	23.44%
vs. ED-O1 P-SW	14.7×
vs. ED-O1 P-HW	6.2×
vs. ED-B P-SW	13.5×
vs. ED-B P-HW	45.3×
vs. ED-B P-ILP	2.2×

Table 3.10: Results of applying an ILP-based HW/SW partitioning (P-ILP) to ED-O2

Task	Device
rgbToGrayscale	HW
convolve2d_smooth	HW
convolve2d_fused	HW
combthreshold	HW
main_begin	SW
main_end	SW
Total exec time (us)	58,873
Total comm. time (us)	10,578
% of comm. time	17.97%
vs. ED-O2 P-SW	7.3×
vs. ED-O2 P-HW	3.5×
vs. ED-B P-SW	7.4×
vs. ED-B P-HW	24.9×
vs. ED-B P-ILP	1.2×

- We declare local buffers inside the hardware tasks, as big as the available size, using the FPGA's BRAMs. This, in turn, imposes a limit on the size of the input image (this is the behavior of the optimized *edgedetect*).

Figure 3.16 shows the execution time for ED-B-D, ED-O1-D, and the two CPU versions of ED-B-D, as measured in the previously described HPC system with an Alveo U250. Furthermore, we also include the execution time for pure software versions, as measured in the CPUs of both our HPC and embedded systems.

Besides the 512×512 image size we use in the static version, we further include images with 5 standardized resolutions (CGA, VGA, WVGA, SVGA, and HD). As we can see, both CPUs far outperform either FPGA version, with the optimized *edgedetect* having a slight edge over the baseline version.

Furthermore, we can compare the execution time of the dynamic ED-O1-D with input size 512×512 to the static version we have for that image size, ED-O1: the former takes 385 ms to execute, while the latter takes 46 ms. This leads to a speedup of 8.4×, which is only achievable by knowing the size of the input.

The very poor scalability of hardware models in the face of dynamic data is, then, something we must contend with. The solution we idealize for this issue is to assign a static size to every dynamic array (or possibly multiple sizes), so that we may at least get conservative estimates of a task's hardware execution time.

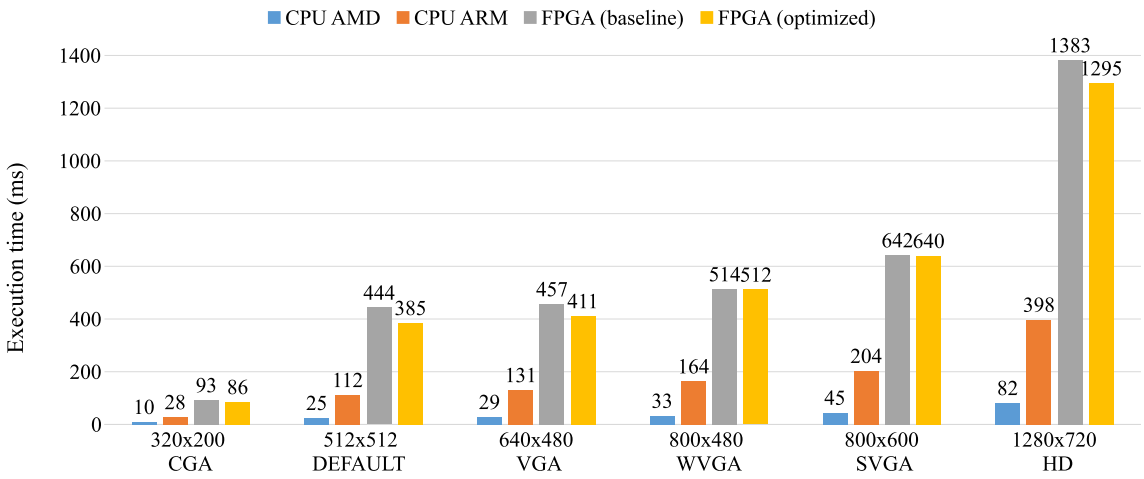


Figure 3.16: Evolution of two dynamic versions of the *edgedetect* application execution time when using different input sizes.

3.7 Summary

This chapter described the methodology adopted, starting with an overview of how we can leverage source-to-source compilation for our purposes. As it is an AST-based approach, we defined a subset of C that reduces the entropy and makes analysis manageable, as well as the transformations necessary for an input application to conform. Next, we formalized our task graph representation, the ETG, which we use alongside the AST to reason about the input program, and to serve as the basis of the partitioning process. This process is guided by a heuristic, greedy algorithm that finds clusters of tasks capturing a preset percentage of the program's equivalent CPU execution time. The heuristic guiding the algorithm is informed by the types of transformations and optimizations that we can apply to the cluster, and analysis is enriched by holistic information from the entire application. Finally, we detailed our evaluation strategy, including relevant benchmarks, and concluded by providing a motivating, handcrafted example of a holistic partitioning.

Having settled on a definition of a task graph suitable for large C applications, and having properly conceptualized and motivated our holistic approach to partitioning and optimizing applications, we will now expand in the following two chapters on a clustering algorithm using the ETG, and on optimizing the code regions matching the achieved clusters.

Chapter 4

Clustering Large Code Regions

This chapter details the heuristic clustering algorithm applied over an ETG. We start by describing non-synthesizable C-language features present in tasks from ETGs obtained from large applications, followed by a set of code transformations that eliminate many of these restrictions. We then focus on the partitioning algorithm, including all the policies that guide expansion. We apply the algorithm to the CortexSuite and *edgedetect* applications, and measure the size and impact of the clusters under different synthesizability conditions.

4.1 Tackling Non-synthesizable C Language Features

HLS, as a process, has several restrictions regarding features and constructs present in high-level languages. While the memory model of C is directly tied to the memory model of a typical Von Neumann CPU, an FPGA, as a non-von Neumann machine, does not preserve the same assumptions. Pointer arithmetic, for instance, is quite limited, as HLS maps every memory access to either a local register, a BRAM/URAM memory port, or an external off-chip access. This makes it difficult to reliably map commonplace C constructs to an HLS design, such as Pointer-to-Pointer arithmetic, dereferencing, and dynamic memory management.

Recursion is another common C feature that is also not feasible on an FPGA. HLS requires each function to have a well-defined resource allocation, which would in turn require the recursive function to have a fixed recursion depth. This is not feasible in most scenarios, and a developer would achieve more success rewriting the code to remove recursion.

Finally, accessing functions with no high-level language implementation or IP module available at compile-time is not possible, as all computations need to be mapped to the FPGA. A subset of these functions are system calls, which are intrinsically tied to the CPU's instruction set, and are not synthesizable even if their source code was provided.

During clustering, some tasks on the ETG may contain one or more non-synthesizable C features, preventing their inclusion in the cluster and limiting its expansion. By synthesizing every task of all 9 CortexSuite-vision applications [137] and the *llama2* application [139] using Vitis HLS, we identify the following 7 non-synthesizable HLS features:

- Present in CortexSuite:
  - Dynamic memory allocations, often present in the form of calls to `malloc()`, `calloc()`, `realloc()`, and `free()`. Allocations performed to create `structs` are particularly tricky, as their conversion into static arrays is made more difficult.
  - Pointer-to-Pointer, a generic case of pointer indirection where a pointer dereferences to another pointer.
  - Pointers to `structs`, which are a subset of pointer indirection caused by pointers to struct field pointers. In CortexSuite, this is exemplified by `struct struct I2D {int width; int height; int *data}`, which is present in every application, and by patterns such as `I2D *foo; foo->data = bar->data;`.
  - `Struct` pointers as arguments, which is similar to the above, but restricted to the specific instance where a `struct` pointer is communicated on a function call.
- Present in *llama2*:
  - Calls to variadic functions, i.e., calls with a varying number of arguments. As an HLS design needs to have a fixed amount of resources, this behavior is not allowed.
  - Function pointers, which prevent the creation of a complete function call graph at compile time.
  - System calls, often hidden behind standard library functions such as `fflush()`.

We focus on tackling the restrictions discovered in these benchmark applications, and consider others, such as recursion, to be out of scope. With this in mind, we propose a set of AST transformations to render these tasks synthesizable.

4.1.1 Synthesizing User-space C Library Function Calls

One of the challenges in adapting C code written for conventional CPUs is dealing with calls to functions from the standard C library. Some of these functions, such as the mathematical functions of `<math.h>`, are widely supported by synthesis tools, and may even be replaced by highly efficient, pre-generated IP modules during synthesis rather than being synthesized from their source.

However, many functions use C language features that are not compatible with HLS. For example, functions `qsort()` and `bsearch()` require a pointer to a comparator function, making them unsuitable for HLS (in addition to the recursivity typically present in `qsort()`). Such functions can be replaced by re-implementations free of these issues, or adapted to inline the comparator function in each call spot.

Furthermore, the implementation of each function may be recursive, which is also not suitable for synthesis. Our solution for these functions is a code transformation that replaces calls to these functions with calls to custom sorting and search functions, suitable for HLS. We deal with

```c
// Before transformations
#include <stdlib.h>
#include <stdio.h>

int compare_ints(const void *a, const void *b) {
    int arg1 = *(const int *)a;
    int arg2 = *(const int *)b;
    return (arg1 > arg2) - (arg1 < arg2);
}

void foo(int *A, int *B, char *str, int n) {
    qsort(A, n, sizeof(int), compare_ints); // sort array A using compare_ints
    int range;
    sscanf(str, "range=%d", &range);   // parse string to get range
    memcpy(B, A, range * sizeof(int)); // copy range elements from A to B, up to range
}

// After transformations
void specialized_qsort(int *base, size_t num) {
    // inlines call to "compare_ints", instead of using a function pointer
}

void specialized_sscanf(const char *str, const char *format, int *range) {
    // specializes parsing for a single integer, rather than using variadic args
}

void foo(int *A, int *B, char *str, int n) {
    specialized_qsort(A, n);
    int range;
    specialized_sscanf(str, "range=%d", &range);
  // transforms memcpy into an explicit loop
    for (int i = 0; i < range; i++) {
        B[i] = A[i];
    }
}
```

Figure 4.1: Example of specialization of C user-space library functions to enable synthesizability.

function pointers by creating an instance of the new sorting/searching function for each call spot, and inline the comparator function in each instance. We show this for `qsort()` in Figure 4.1. The recursivity issue is dealt with by providing implementations that are not only free of recursion, but also exploit the FPGA's parallelism: sorting is done using insertion sort, as shifting an array can be a one-cycle operation on a reconfigurable fabric; and search is achieved through binary search.

Other C library functions, such as the string-parsing function `sscanf(char* str, char* format, ...)`, rely on variadic arguments, which are not synthesizable. Nevertheless, as the type and number of arguments in each call spot are known at compile time, each instance of these functions can be specialized to avoid variadic arguments. In Figure 4.1, the call to `sscanf()` is specialized to parse a single integer token.

Finally, there are some C library functions, such as `memset()` and `memcpy()`, that are accepted by synthesis tools (e.g., Vitis HLS), but that still hide implementation details. By replacing these functions with simple functionally equivalent for-loops, we expose simpler code structures that can be optimized by, e.g., loop pipelining, rather than having a bottleneck caused by a function call that was not inlined. We show an example of this for `memcpy()` in Figure 4.1.

4.1.2 Asynchronous Host Execution of Kernel-space Function Calls

The previous transformations are possible for functions that are part of the C standard and that operate exclusively in user space (i.e., perform no system calls). However, many C library functions rely on system calls, which need to be host-side executed on a typical CPU-FPGA system. Figure 4.2 (a) shows the lifetime of a CPU-FPGA application that needs to explicitly return to the host in order to execute calls to `printf()` and `assert()`. This is inefficient, as it requires us to split the code region into three separate FPGA kernels which may incur additional communication costs.

Many of these functions, such as `fopen()` or `rand()`, have return values required by the code region, which needs to be split in order to keep the call host-side. However, many other functions either have no return value, or have return values unused by the caller. This is the case with both calls used in the example of Figure 4.2.

We propose a scheme for executing functions that rely on system calls without or with unused return values, as shown in Figure 4.2 (b). In this scheme, whenever the FPGA kernel needs to call one of these functions, it instead writes the function arguments to an off-chip buffer in shared memory, and continues executing without acknowledging the return value (i.e., a fire-and-forget mechanism, where the FPGA assumes that the call is executed by the CPU). The CPU host, on the other hand, constantly polls the buffers in program order, waiting for function arguments, and executes each function once they are available. We call this mechanism "asynchronous host calls".

Similarly to the ETG, we have an asynchronous host call per call site. In the current version, each call site has its own shared buffer, created according to the size of the arguments in the call and the number of times the call spot is executed. E.g., if a call spot prints a 32-bit integer using `printf()`, and is executed inside a loop with 64 iterations, the buffer size is 256 bytes. This ensures communication is asynchronous and non-blocking, with the FPGA triggering calls while the host polls the buffer at its own rate. It is also the responsibility of the FPGA to signal the host when the buffer is closed (i.e., after the call spot is called for the last time).

For each asynchronous host call, we require two implementations of the asynchronous function: one for the host and the other for the FPGA code region. The compiler is responsible for modifying the code and generating the required implementations. On the FPGA side, it ensures that the right arguments are written to the host using appropriate sizes; and on the host, it ensures that data is read in adequate chunks, and that the original call is executed using that data. Furthermore, all constant arguments (e.g., the format string in `printf("%d", i)`) are kept on the host implementation, and the FPGA communicates only what is strictly needed.

We further note that this approach can also be applied to user-space functions with no available implementation, as long as they do not have an acknowledged return value.

4.1.3 Flattening Structs to Simplify Pointers

One challenge in synthesizing large code regions resides in dealing with structs. Many structs, such as the one in Figure 4.3 (present in most CortexSuite [137] applications), lead to indirect

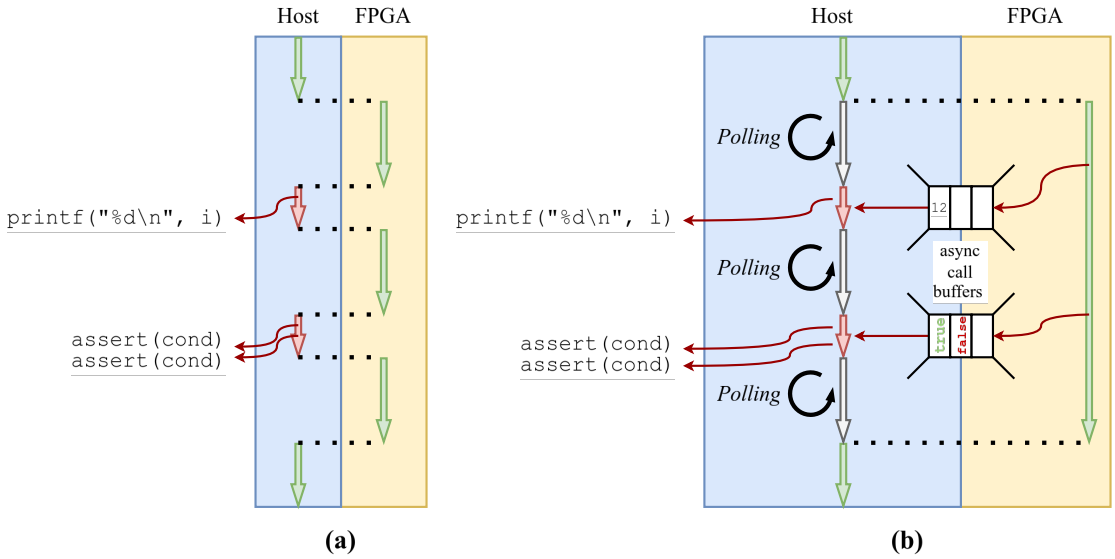


Figure 4.2: Lifetime of a CPU-FPGA application that calls `printf()` and `assert()` twice. In scenario (a), the FPGA code region is split into 3 separate kernels, reverting to the host in order to execute the calls; in scenario (b), the FPGA region is a single kernel, which executes the calls asynchronously on the host.

pointers if one or more of their fields are pointers, e.g., `grayImg->data`. However, in many cases, these limitations can be overcome by flattening a struct into local variables representing each of its fields. Scalar fields become scalar variables, and pointer fields become pointer variables that can be initialized by a single memory allocation. This is shown in Figure 4.3. Flattening structs is a complex operation, as it requires the modification of all references to the struct and to its fields, whether in arithmetic expressions, function calls/signatures, and assignments.

We apply struct flattening for all structs in the program, provided they follow two conditions: they must be standalone variables, i.e., not part of an array of structs, and they can have, at most, one dynamically-allocated pointer field. This latter condition is exemplified in Figure 4.3, in the flattening of the original `malloc()` call into three separate variables. The size of the allocation is given by `msize`. If we have a single dynamically-allocated field, we can obtain its size by subtracting the size of the scalar fields. If we had more than one dynamically-allocated field, we could still find their collective size, but not the size of each individual field. This example also illustrates how scalar fields do not need to be dynamically allocated, further reducing the need for `malloc()` calls.

4.1.4 Hoisting Dynamic Memory Allocations

One major challenge towards synthesizability is the presence of memory allocations (e.g., calls to `malloc()` or `calloc()`) in the code region of interest. We include two distinct transformations for dealing with these when all memory allocations in the region are a simple assignment to a

```c
// Struct for an image, obtained from CortexSuite
typedef struct {
    int width;
    int height;
    int *data;
} I2D;

// Non-synthesizable function due to struct pointers
void create_gray_from_color(I2D *colorImg, I2D **result) {
    I2D *grayImg = malloc(sizeof(I2D) + colorImg->width * colorImg->height * sizeof(int));
    fill_base_color(grayImg, 0x111111);

    for (int i = 0; i < colorImg->height; i++) {
        for (int j = 0; j < colorImg->width; j++) {
            int idx = i * colorImg->width + j;
            int gray_val = to_gray(colorImg->data[idx] + 1);
            grayImg->data[idx] = grayImg->data[idx] + gray_val;
        }
    }
    *result = grayImg;
}

// Synthesizable function obtained after struct flattening
void create_gray_from_color(int colorImg_width, int colorImg_height, int *colorImg_data,
                            int *result_width, int *result_height, int **result_data) {
    I2D *grayImg = (I2D *)malloc(colorImg_width * colorImg_height * sizeof(int));
    int grayImg_width = colorImg_width;
    int grayImg_height = colorImg_height;
  size_t msize = sizeof(I2D) + colorImg->width * colorImg->height * sizeof(int);
    int *grayImg_data = (int *)malloc(msize - sizeof(int) - sizeof(int));

    fill_base_color(grayImg_width, grayImg_height, grayImg_data, 0x111111);

    for (int i = 0; i < colorImg_height; i++) {
        for (int j = 0; j < colorImg_width; j++) {
            int idx = i * colorImg_width + j;
            int gray_val = to_gray(colorImg_data[idx] + 1);
            grayImg_data[idx] = grayImg_data[idx] + gray_val;
        }
    }
    *result_width = grayImg_width;
    *result_height = grayImg_height;
    *result_data = grayImg_data;
}
```

Figure 4.3: Enabling the synthesizability of a C function with struct pointers by applying struct flattening.

pointer. This is ensured by the previous transformations, as well as by the transformations required to generate an ETG.

The first transformation was already mentioned: if the memory allocation uses values known at compile time, the compiler optionally turns it into a static array, assuming its size never changes. Turning a dynamic array into a static one allows its storage in on-chip memories at the expense of using more FPGA resources, in particular block RAMs (BRAMs), but with the advantage of reduced memory access latency compared to off-chip memories. The decision about transforming into static arrays, therefore, falls on either the developer using the transformation, the target platform's requirements, or the heuristics used by an automatic HW/SW partitioning process.

The second transformation is more general, and it involves moving the memory allocation to

```
1 void sum_arrays(int *A, int *B, int *C, int size) {
2     for (int i = 0; i < size; i++) {
3         A[i] = B[i] + C[i];
4     }
5 }
6
7 void power_arr(int *A, int *D, int power, int size) {
8     for (int i = 0; i < size; i++) {
9         for (int j = 0; j < power; j++) {
10            D[i] *= A[i];
11        }
12    }
13 }
14
15 void print_power(int *A, int *B, int *C, int n, int size) {
16    int *D = calloc(size, sizeof(int));
17    power_arr(A, D, n, size);
18    printf("N = %d: D = {%d, ..., %d} = %d\n", n, D[0], D[size - 1]);
19    free(D);
20 }
21
22 void entrypoint(int *A, int *B, int *C, int size) {
23    sum_arrays(A, B, C, size);
24    for (int n = 0; n < 4; n++) {
25        print_power(A, B, C, n, size);
26    }
27 }
```

Figure 4.4: Source code matching the ETG in Figure 4.5, showing a non-synthesizable code region hampered by dynamic memory allocations and calls to `printf()`.

a location outside the code region. This is shown in Figure 4.4: if we assume our code region of interest is `print_power`, we notice that it is not currently synthesizable from Figure 4.5 (a), as it has a call to `calloc()` and `free()`. In Figure 4.5 (b), however, we raise the hierarchy of these calls, moving them out of the region and communicating only the pointer. Raising the hierarchy on the ETG is reflected in the AST as moving the call site into the caller of its current function, and adding the pointer as an additional parameter of the callee. This transformation also requires an analysis of the pointer's lifetime using the ETG, as we need to traverse the tasks that use the pointer and determine whether its memory region is replaced or resized by another memory allocation, or deleted by a call to `free()`. We cannot deal with the former, but the latter is solved by moving the call to `free()` immediately after the region, at the same scope as the memory allocation that originated it.

By applying the asynchronous call mechanism, in Figure 4.5 (c), we can enable the synthesis of `print_power`. Iteratively applying `calloc()` hoisting may enable us to go beyond the region of interest, as shown in Figure 4.5 (d), by including the sibling task `sum_arrays` and raising the level of hierarchy up to `entrypoint`.

4.1.5 Recursively Inlining a Code Region

The final transformation is to recursively inline every function call in the region, so that only one large function remains (i.e., apply task fusion to every task in the cluster). This allows us to extend

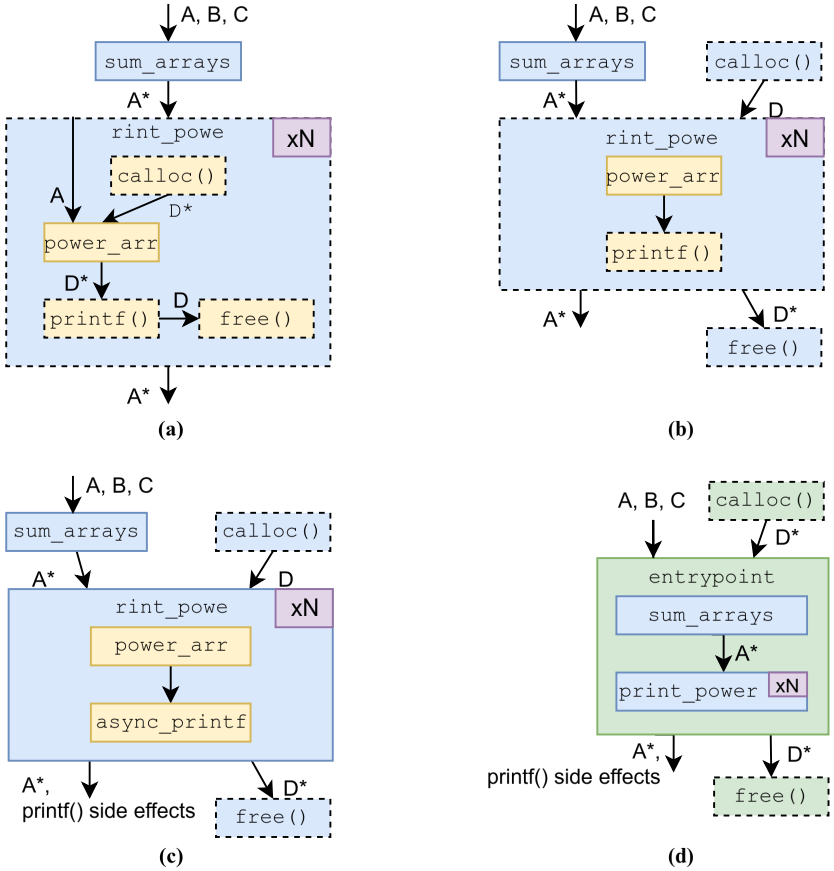


Figure 4.5: Extended Task Graphs (ETGs) showing the transformations applied to make the example synthesizable. Tasks with dashed borders are not synthesizable, and '*' represent data items modified by the previous task. (a) shows the original ETG; (b) shows a transformation where `calloc()` and `free()` are moved in the hierarchy; (c) shows a conversion of `print()` into a synthesizable code that triggers the asynchronous version (offloaded to CPU); and (d) further increases the hierarchy of `calloc()` and `free()` to move them completely out of the region, making `region_wrapper` completely synthesizable.

`malloc()` hoisting to any memory allocation in the region (provided it is not reallocated). Inlining every call also allows for the simplification of pointer arithmetic, solving limitations caused by `struct` pointers as call arguments. The final function also enables new opportunities for constant folding and propagation, further simplifying the source code.

4.1.6 Summary of Transformations

Table 4.1 summarizes, for each HLS restriction identified, the set of transformations that solve it. We note that these transformations are only applied on the final cluster obtained by the algorithm. The algorithm decides, based on the transformations available, whether it can add a non-synthesizable task to the cluster.

Table 4.1: Summary of non-synthesizable C constructs, and the source-to-source transformations that lift them

Non-synthesizable C construct	Transformations
Dynamic memory allocations	`malloc()` hoisting + region inlining
Pointer-to-Pointer	Region inlining
`Struct` Pointers	`Struct` flattening
`Struct` Pointers as Arguments	`Struct` flattening + region inlining
Calls to variadic functions	Function specialization for each call site
Function pointers	Inlining the call to the function pointer
No-return System calls	Asynchronous host calls

4.2 A Heuristic Greedy Algorithm for Clustering

To create a large cluster, we use Algorithm 1. The first step is determining the *Goal* of the algorithm, i.e., the metric that the algorithm tries to reach or approach, so that $Value(cluster) >= Goal$. We propose two metrics, based on the properties of the ETG described in Chapter 3:

- **Computation percentage (%Comp)**: as each task is annotated with its percentage of CPU computation time, obtained through profiling, we can define *Goal* as the percentage of computation that the cluster should encompass. E.g., we want to find a cluster that represents at least 80% of the program's execution time. Although some variability is expected due to profiling imprecisions, the ETG structure and underlying C guarantee that $Value(T) \approx \sum_{x \in S} Value(x)$, for a set S of all subtasks of T. In other words, the percentage of computation of a task is approximately equal to the sum of the computation percentage of its subtasks, at a lower level of hierarchy.

- **Maximum latency**: this goal uses the estimated HLS latency of a cluster as the metric. We define a maximum latency allowed in a cluster, and the algorithm expands until it can no longer add more tasks without exceeding the limit. While every task is annotated with an HLS latency estimation, $Value(T) \neq \sum_{x \in S} Value(x)$, as the sum of subtask latencies is not always equal to the latency of the parent task. HLS tools can explore further optimizations when provided with multiple call sites, particularly if inlining is applied. Therefore, the algorithm needs to synthesize the cluster every time it tests if it should add a new task.

The algorithm is a bottom-up greedy algorithm that builds several candidate clusters to select the best one. Each cluster starts with a leaf task, i.e., tasks that do not encapsulate any other subgraphs at a lower hierarchical level. If the leaf task is a valid task for offloading, each of its sibling tasks, at the same hierarchical level, is tested for cluster inclusion:

- **Synthesizability check**: if the task is not synthesizable, the algorithm can only allow it if there is a transformation available capable of overcoming its restrictions. As all transformations are only applied to the final cluster, and not during the algorithm, we can configure the synthesizability conditions to only allow some non-synthesizable features, e.g., we may

Algorithm 1 Heuristic Greedy Clustering Algorithm

```
1:  function FINDBESTCLUSTER(ETG, Goal)
2:      candidates ← ∅
3:      leaf_tasks ← FINDLEAFTASKS(ETG)
4:      for all T_f ∈ leaf_tasks do
5:          candidates ← candidates ∪ {CREATECLUSTER(T_f, Goal)}
6:      end for
7:      chosen_cluster ← max(candidates)
8:      return chosen_cluster
9:  end function

10: function CREATECLUSTER(T, Goal)
11:     if ¬SYNTHESIZABLE(T) then
12:         return ∅
13:     end if
14:     cluster ← {T}
15:     if VALUE(cluster) ≥ Goal ∨ T.is_root then
16:         return cluster
17:     end if
18:     for all siblings T_s of T do
19:         if SYNTHESIZABLE(T_s) ∧ DESIRABLE(T_s, cluster) then
20:             cluster ← cluster ∪ {T_s}
21:         end if
22:     end for
23:     if |cluster| = #siblings(T) + 1 then
24:         return CREATECLUSTER(T.parent, Goal)
25:     else
26:         return cluster
27:     end if
28: end function
```

want to allow tasks with struct pointers, but not tasks with system calls. The more features we allow, the more transformations are applied on the final cluster, raising the potential for constructive and destructive interference.

- **Desirability check**: by default, a task is considered desirable if it can be synthesized. However, we can optionally enrich the decision of accepting or rejecting the task by providing additional information. These consist of additional heuristics obtained through a holistic analysis of the program, e.g., reject a data-intensive task with high memory access costs but few computations.

If both a leaf task and all of its siblings are accepted into the cluster, the algorithm replaces these tasks in the cluster by their parent, i.e., the task at a higher level of hierarchy on the ETG. The process is then repeated for that task's siblings, increasing the level of hierarchy with each iteration. We stop expanding the cluster when we meet one of three conditions:

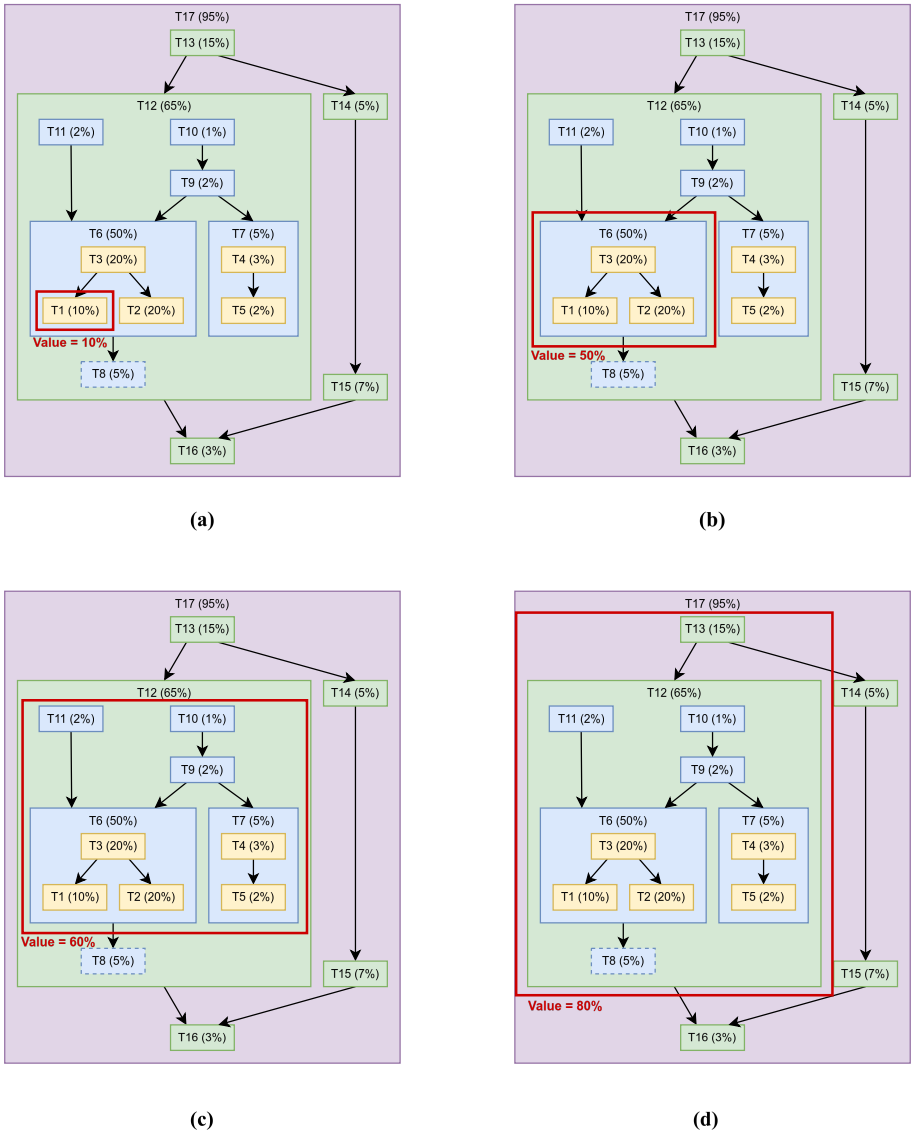


Figure 4.6: Stages of the clustering algorithm for an example ETG, from the starting leaf task T1 (a), to expanding towards its siblings (b), to dealing with a non-synthesizable task (c), and finally reaching its goal (d).

- If there is one or more sibling tasks that are not accepted, expansion stops, as the level of hierarchy can no longer be raised.
- If the target goal is reached, i.e., $Value(cluster) >= Goal$.
- If the root of the ETG is reached.

If there is a single task in the ETG that maps neatly into *Goal* (e.g., there is a single task at a high hierarchical level representing 78% of the computation time for $Goal = 80\%$), we can use it as the root of a new, smaller ETG. In this case, if the algorithm is capable of expanding up to the root, the cluster equals the ETG.

In Figure 4.6, we show an example of the algorithm applied to an ETG with 17 tasks and 4 levels of hierarchy, representing 95% of the CPU computation time. We configure the algorithm with $Goal = 80\%$. As this ETG has 13 leaf tasks, it would create up to 13 candidate clusters. The example shows the expansion process starting from leaf task $T1$ (Figure 4.6 (a)). The current cluster is $C = \{T1\}$. Next, it tests its sibling tasks, $T2$ and $T3$, and successfully adds them to the cluster, leading to cluster $C = \{T1, T2, T3\}$. As all these tasks are the subtasks of $T6$, we raise the level of hierarchy and set the cluster to $C = \{T6\}$. The process repeats for the siblings of $T6$, successfully adding tasks $T9$, $T10$, and $T11$ (Figure 4.6 (b)). However, as task $T8$ is not synthesizable, the synthesizability check needs to verify if there is a transformation available capable of lifting the restriction (Figure 4.6 (c)). The check passes, and $T8$ is included in the cluster. Once again, as the cluster consists of all subtasks of $T12$, the cluster is updated to $C = \{T12\}$ (Figure 4.6 (d)). As the current cluster represents 65% of the computation value, it needs only add one additional sibling task representing 15%, $T13$, to reach the goal of 80%. Expansion stops once the goal is reached, even if more tasks could have been added.

4.3 Measuring the Impact of Transformations

In this section, we study the impact of the proposed transformations across two dimensions. Firstly, we provide three use cases to explore their impact on the final Place-and-Routed HLS designs. Then, using the proposed clustering algorithm and the CortexSuite applications [137], we explore how enabling/disabling transformations promotes or hinders the creation of large clusters for offloading. We recall that all synthesis is done using Vitis HLS, and targeting the FPGA of the ZCU102 board.

4.3.1 Impact of Asynchronous Host Calls

In order to measure both the impact and usefulness of the asynchronous call mechanism, we perform two case studies: we apply it to the *llama2* application [139], which is impacted by calls to the standard output, and to the *spam-filter* application of Rosetta [143] to showcase the potential of using this mechanism to debug HLS designs using a software-centric approach.

Standard output on an FPGA code region

As described in Chapter 3, *llama2* is an LLM inference application, generating a sequence of text for a given prompt and a given pre-trained model. As with most LLMs, it generates words one at a time, which is reflected in the output that the user sees. In this specific implementation, *llama2* outputs the generated word using `printf()`, for a total of 256 words. Accelerating LLMs in hardware is increasingly relevant, and being able to synthesize the entire *llama2* without compromising on the output's behavior is paramount, particularly when considering that we could replace

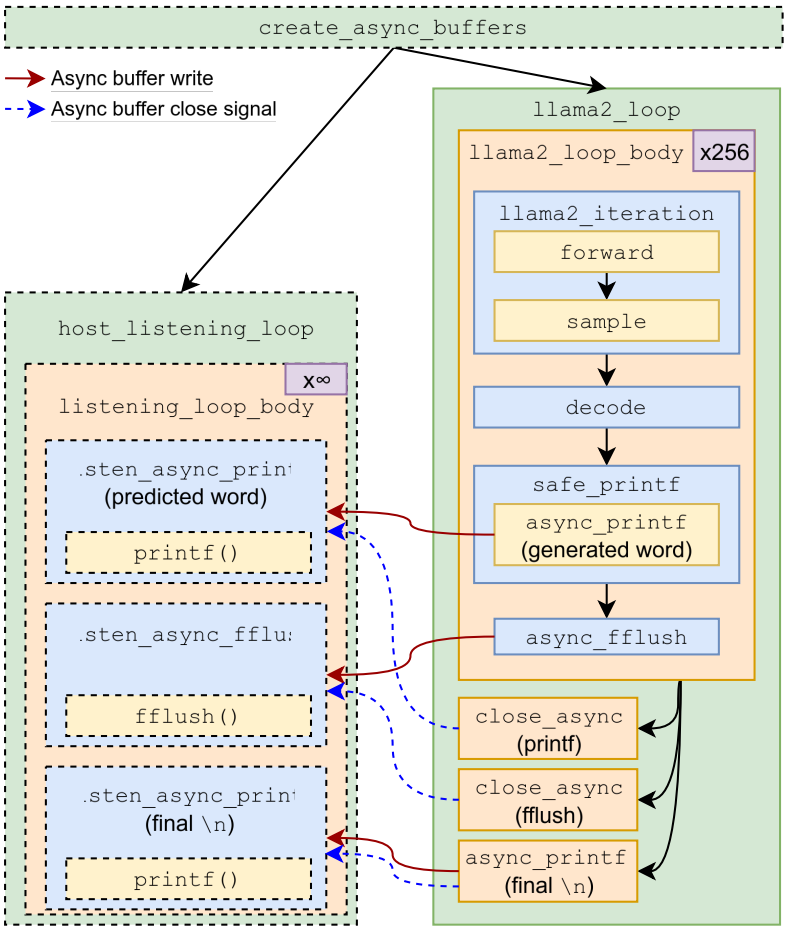


Figure 4.7: Extended Task Graph (ETG) of *llama2*, after applying a transformation to enable asynchronous host calls. Dashed tasks are not synthesizable and represent the host component that runs alongside the FPGA kernel. Tasks `forward`, `sample`, and `decode` have further subtasks down their hierarchy, which are omitted for simplicity.

`printf()` by some other output stream, such as a socket on a distributed environment. In Figure 4.7, we show the ETG for *llama2*. The code region of interest is task `llama2_loop` and its subtasks.

However, before dealing with `printf()`, we need to apply transformations to turn other parts of *llama2* synthesizable. In particular, *llama2* uses `qsort()` to sort a list of tokens, which the compiler replaces with a synthesizable implementation. Furthermore, it also uses `sscanf()` to sanitize some tokens using a format string, which is specialized into an implementation without variadic arguments. Finally, our compiler substitutes/inlines all calls to `memset()` and `memcpy()`.

These transformations enable the HLS of the main hotspot, the `llama2_iteration` task, as depicted in Figure 4.7 and shown in Table 4.2 (version L1). We also show the synthesis results on this table, for a single invocation of this task. This benchmark relies on dynamic data with few bounds known at compile time, and so we determine the number of iterations of each loop through CPU profiling, using a pre-trained model with 15 million parameters. We then annotate every loop with its minimum, average, and maximum number of iterations, allowing Vitis HLS to obtain a

Table 4.2: Versions of the *llama2* application depicting the offloaded code regions, latency, and HLS resource usage post Place-and-Route

Version	Code Region	#Stmt	#Tasks (%)	CPL	Latency (s)	BRAM (%)	DSP (%)	FF (%)	LUT (%)
L1	`llama_iteration`	650	35 (73%)	22	44.19	5 (0.27%)	84 (3.33%)	31,473 (5.74%)	28,575 (5.21%)
L2	`llama_loop` (async disabled)	804	48 (100%)	34	11,313.43	5 (0.27%)	84 (3.33%)	31,759 (5.79%)	28,862 (5.27%)
L3	`llama_loop`	837	51 (100%)	37	11,313.44	9 (0.49%)	84 (3.33%)	41,981 (7.66%)	37,442 (6.83%)

#Stmt number of statements, *#Tasks (%)* number of offloaded tasks (percentage of total tasks), *CPL* ETG critical path length, *BRAM* block random access memory, *DSP* digital signal processor, *FF* flip-flop, *LUT* lookup table

cycle count. The BRAM usage is minimal, as all dynamic data is accessed through DMA. Note that instrumenting every loop with an iteration counter, executing the program with a given input, and annotating it with the profiling results are also automated by our compiler.

Regarding the latency, we need to account for all 256 invocations, resulting in a total latency of $44.19 \times 256 = 11312.64$ s, without including the communication costs and overhead of each invocation. Although this latency is too high to be useful, all HLS designs are unoptimized besides the default Vitis HLS transformations. While we recognize several avenues for optimization, as we are focused only on synthesizability, we consider them out of this paper's scope.

At last, the compiler turns the `llama2_loop` task and its subtasks synthesizable by applying asynchronous host calls transformations. Not only is there one `printf()` call spot for when a new word is generated, there is also a call to `fflush()` that immediately follows it, as well as one final `printf()` to output a newline character. Both the `fflush()` and the second `printf()` call spots require only a buffer for one byte, as both their arguments can be kept host-side. The first `printf()`, on the other hand, is called 256 times, and outputs an entire string through the buffer, representing the word generated by the LLM. Knowledge of the application shows that the maximum word size is 29 characters, and so we can configure a conservative buffer size of $256 \times 29 = 7{,}424$ bytes. On the host side, the compiler adds a new task called `host_listening_loop`. This task polls all asynchronous call buffers in a loop, dispatching the host-side calls whenever new data is detected, and only ends when all buffers are closed by the FPGA. The buffers of the first `printf()` and `fflush()` need to be explicitly closed, represented by the `close_async` tasks in Figure 4.7. Furthermore, as seen in Table 4.2, synthesizing all possible tasks in the ETG leads to a critical path of 37 tasks (for a total of 51). This informs us that, unlike *disparity* and *spam-filter*, we have a degree of task-level parallelism that can be exploited by future HW/SW partitioning and optimization algorithms.

We synthesize this version of *llama2* and compare it to a version with all `printf()` and `fflush()` calls removed, to gauge the impact of adding this mechanism. These versions, L3 and L2 respectively, are shown in Table 4.2, where we can see that they capture all *llama2* ETG tasks. We notice an increase in all resources (+0.29% BRAMs, +1.87% FFs, +1.56% LUTs) except DSPs, which is expected considering this mechanism does no arithmetic besides simple

```
1 void SgdLR_sw(...,
2 @   int8_t *printf_buf, async_kernel_info *printf_info,
3 @   int8_t *assert0_buf, async_kernel_info *assert0_info,
4 @   int8_t *assert1_buf, async_kernel_info *assert1_info    ) {
5
6     for (epoch = 0; epoch < EPOCHS; epoch++) {
7         for (id = 0; id < N; id++) {
8             dot = dotProduct(...);
9         prob = Sigmoid(dot);
10            computeGradient(...);
11            updateParameter(...);
12        }
13        // original call:
14        // printf("Epoch: %d\n", epoch);
15 @      int64_t printf_args[1] = {epoch};
16 @      async_printf(printf_buf, printf_info, false, printf_args, 1);
17    }
18 @   close_async(printf_info);
19    bool c0 = fabs(theta[0] - THETA0) <= ERR;
20    bool c1 = fabs(theta[1023] - THETA1023) <= ERR;
21    // assert(c0); // original assert() calls
22    // assert(c1);
23 @   async_assert(assert0_buf, assert0_info, true, c0);
24 @   async_assert(assert1_buf, assert1_info, true, c1);
25 }
```

Figure 4.8: Code of the *spam-filter* benchmark with the asynchronous call mechanism applied to enable debugging.

additions. The 11,313.44 s of latency is also similar to the 11,312.64 s of latency of the smaller kernel invoked 256 times, with the further advantage of not incurring the overhead associated with 256 invocations.

Debugging with *printf()* and *assert()*

Our second case study uses the *spam-filter* benchmark from Rosetta [143], a suite made for use in HLS, that presents no challenges regarding synthesizability, as shown by version S1 of Table 4.3. The benchmark implements a stochastic descent algorithm to train a linear regression for detecting email spam. Training is done by traversing the training dataset and calculating the gradient at each point. Multiple traversals are necessary, as specified by the number of epochs.

We use it to showcase a debugging scenario, where a developer wants to run the benchmark for N epochs, printing a message to the standard output via `printf()` every time a new epoch begins to indicate that the program is progressing. Furthermore, we introduce two `assert()` statements to verify whether two values in the trained model have acceptable precision. We describe this version in Table 4.3 (version S2). We further note that `assert()` is not a function, but a macro that resolves to a conditional check and an `exit()` system call. Nevertheless, our compiler extension can generate an asynchronous call to `assert()` as if it behaved like a function, and it also adds additional code to ensure that the kernel is terminated before the call to `exit()`. Moreover, as Clava uses Clang's preprocessor, all macros are resolved before generating the AST, exposing any abstracted function calls.

Table 4.3: Versions of the *spam-filter* benchmark depicting the offloaded code regions, latency, and HLS resource usage post Place-and-Route

Version	Description	#Stmt	#Tasks (%)	CPL	Latency (s)	BRAM (%)	DSP (%)	FF (%)	LUT (%)
S1	No async calls	79	7 (100%)	7	16.50	33 (1.81%)	12 (0.48%)	11,450 (2.09%)	6,208 (1.13%)
S2	With `printf()` + `assert()`	101	10 (100%)	10	17.62	33 (1.81%)	12 (0.48%)	19,028 (3.47%)	11,974 (2.18%)

#Stmt number of statements, *#Tasks (%)* number of offloaded tasks (percentage of total tasks), *CPL* ETG critical path length, *BRAM* block random access memory, *DSP* digital signal processor, *FF* flip-flop, *LUT* lookup table

Applying our asynchronous host call transformation to this code results in the code in Figure 4.8. Each call spot is replaced by an asynchronous call, which requires three arguments regardless of the underlying function: a dedicated buffer to hold the data, a struct holding the information regarding the buffer's status, and a flag to signal whether this call closes the buffer. One important detail is that when we have an asynchronous call that repeats, we do not know when the last call is done, and so we can never close the buffer when a call occurs. Instead, we must do it explicitly at a higher scope level, after the repetitions, using a dedicated function. This happens with the call to `async_printf()` in Figure 4.8, as it is inside a loop. Its buffer is closed later by calling `close_async()`. Calls to `async_assert()`, are executed only once, and we can close the buffer immediately in the call itself.

To evaluate the impact of these async calls for debugging in the resulting HDL design, we synthesize the application, both with and without async calls. We show the Vitis HLS synthesis results in Table 4.3. Their footprint is minimal (requiring only 1.38% more FFs and 1.05% more LUTs, at a small latency increase from 16.50 s to 17.62 s), making them suitable for both debugging and production scenarios. We validate the mechanism's functionality through a HW/SW co-simulation, and implement the host communication and handling of async calls using the Xilinx Runtime (XRT). The co-simulation is thorough, emulating the entire PetaLinux OS [155]. Finally, we run the hybrid application on the ZCU102 board and detect no measurable difference in the execution time of the CPU-FPGA application, with and without async calls.

4.3.2 Impact of Hoisting Dynamic Memory Allocation Calls

The third and final case study shows the impact of hoisting calls to `malloc()`, as well as exploring the potential of turning a `malloc()` into a static array. The *disparity* application, from CortexSuite, calculates the apparent motion between a pair of stereo 32-bit RGB images of size 587×480 (1.12 MB per image), and in the process, it uses 6 buffers to hold intermediate results. These buffers have sizes at least equal to that of the input images, with some having up to eight extra rows and columns (1.16 MB per buffer). Figure 4.9 shows the ETG for this application, after applying function outlining. As shown, it calculates the output image `retDisp` by executing the `correlateSAD_2D` and `findDisparity` tasks a total of 64 times. However, this code region

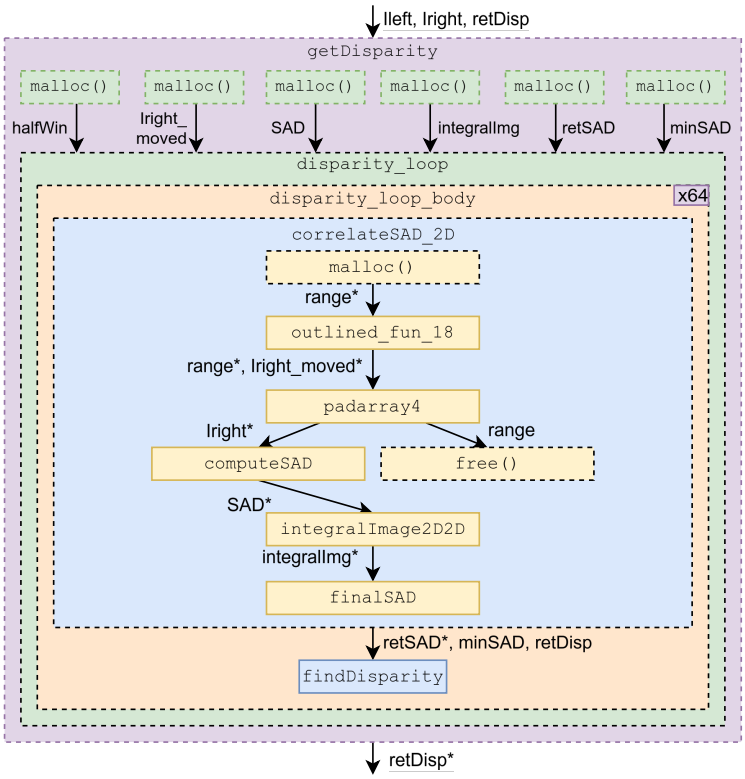


Figure 4.9: ETG of the *disparity* benchmark. Tasks with dashed borders are not synthesizable.

cannot be synthesized, as `correlateSAD_2D` has two non-synthesizable tasks, a `malloc()` and a `free()`. This limits the size of the synthesizable sub-region to the sequence of 5 tasks, starting with `outlined_fun_18` and ending with `finalSAD` (version D1 in Table 4.4). Furthermore, we would need to split the region into two separate FPGA kernels, as `findDisparity` is synthesizable (version D2 in Table 4.4). The resulting designs have a latency of 0.43 s and 0.20 s, respectively. Considering that both sub-regions need to be executed 64 times each, they result in a total of $(0.43{+}0.20) \times 64 = 40.32\ s$ of FPGA latency, without considering the overhead of communicating data 128 times.

Our compiler employs the `malloc()` hoisting transformation to turn `correlateSAD_2D` synthesizable, and consequently all tasks in the region up to `disparity_loop`. In this case, the memory allocation is done using a size known at compile time, allocating only 8 bytes of space. This can be turned into a static array with negligible impact on resource usage, as the compiler analysis of the ETG shows us that the array is never resized. The call to `free()` can also be removed, resulting in version D3 of Table 4.4. Resource usage is comparable to that used by the 5-task sequence and `findDisparity` together. The estimated latency of 40.76 s is also comparable to the 40.32 s it takes for the 128 kernel invocations, except in this scenario, we only have a single invocation, incurring communication costs and accelerator overhead only once instead of 128 times.

No designs undergo any additional optimizations besides the loop unrolling/pipelining done

Table 4.4: Versions of the CortexSuite *disparity* benchmark depicting the offloaded code regions, latency, and HLS resource usage post Place-and-Route

Version	Code Region	#Stmt	#Tasks (%)	CPL	Latency (s)	BRAM (%)	DSP (%)	FF (%)	LUT (%)
D1	5-task sequence	153	5 (42%)	5	0.43	4 (0.22%)	44 (1.75%)	10,490 (1.91%)	9,508 (1.73%)
D2	`findDisparity`	27	1 (8%)	1	0.20	2 (0.11%)	0 (0.00%)	3,766 (0.69%)	3,246 (0.59%)
D3	`disparity_loop`	217	11 (92%)	11	40.76	4 (0.22%)	44 (1.75%)	12,618 (2.30%)	10,814 (1.97%)
D4	`disparity_loop` + static `retSAD`	218	11 (92%)	11	40.31	518 (28.40%)	45 (1.79%)	10,501 (1.92%)	9,197 (1.68%)

#Stmt number of statements, *#Tasks (%)* number of offloaded tasks (percentage of total tasks), *CPL* ETG critical path length, *BRAM* block random access memory, *DSP* digital signal processor, *FF* flip-flop, *LUT* lookup table

by default in Vitis HLS, but the larger design offers us more potential for optimizing, e.g., we can inline all functions and have a single pipeline in the outer loop, which is impossible with split kernels. We could further optimize the memory usage of the region, as there are at least two similarly-sized buffers, `SAD` and `minSAD`, that never overlap (i.e., the last read/write to `SAD` happens before the first read/write to `minSAD`). This means that both buffers could share the same memory space, be it in DRAM or in BRAMs.

Finally, we consider the 6 memory allocations present in the top-level task `getDisparity`. They range from 1.12 MB to 1.16 MB, and are unsuitable for an FPGA with a modest amount of BRAMs. To show this scenario, we select one of these, `retSAD`, lower its hierarchy into `disparity_loop`, and turn it into a static array (version D4 of Table 4.4). The synthesis results show that while latency improved slightly, the BRAM usage went from 0.22% to 28.40%. This suggests that there is potential in turning more memory allocations into static arrays, and we expand on this idea in Chapter 5.

4.3.3 Measuring the Interference Between Transformations

In this experiment, we study how different non-synthesizable constructs impact the clustering of tasks, so that we try to find the largest possible cluster in terms of percentage of computation using Algorithm 1. We generate ETGs for all 9 CortexSuite and the *edgedetect* applications, and define the goal as capturing 80% of the computation percentage.

In Table 4.5, we show the clusters we can achieve, in terms of the number of tasks and their computation percentage, by lifting different HLS restrictions on each run, i.e., by enabling/disabling the code transformations that can be applied over the resulting cluster. We start with a baseline cluster, where we do not lift any HLS restriction. Then, we create clusters by lifting a single non-synthesizable construct, for every construct present in these applications (memory allocations, pointer-to-pointer, structs with pointers, and struct arguments with pointers). Finally, we create clusters by lifting all non-synthesizable constructs.

Table 4.5: Largest Clusters Achieved by Lifting HLS Restrictions (*Goal* = 80%)

Application	None lifted	Malloc	Pointer to Pointer	Struct Pointers	Struct Arg with Pointer	All lifted
disparity	1 (0.0%)	1 (0.0%)	1 (0.0%)	4 (76.8%)	1 (0.0%)	12 (88.6%)
localization	1 (0.0%)	1 (0.0%)	1 (0.0%)	1 (0.6%)	1 (0.0%)	486 (81.9%)
mser	1 (0.0%)	1 (0.0%)	11 (29.5%)	1 (50.2%)	1 (5.4%)	57 (94.0%)
multi-ncut	1 (0.0%)	1 (0.0%)	1 (0.0%)	1 (99.9%)	1 (0.0%)	21 (99.9%)
sift	1 (0.1%)	1 (0.1%)	1 (4.5%)	1 (2.4%)	1 (0.1%)	562 (98.0%)
stitch	1 (0.0%)	1 (0.0%)	2 (34.7%)	1 (15.0%)	1 (0.0%)	130 (97.6%)
svm	1 (0.0%)	1 (0.0%)	1 (0.0%)	1 (0.0%)	1 (0.0%)	2,021 (73.6%)
texture-syn	6 (73.0%)	6 (73.0%)	6 (73.0%)	6 (73.0%)	6 (73.0%)	6 (73.0%)
tracking	2 (3.4%)	2 (3.4%)	2 (3.4%)	1 (24.3%)	2 (3.4%)	368 (78.6%)
edgedetect	7 (76.0%)	7 (76.0%)	7 (76.0%)	7 (76.0%)	7 (76.0%)	7 (76.0%)

Taking *disparity* as an example, the baseline cluster has only a single task, with negligible execution time. This shows that, without lifting any restriction, we cannot find a cluster interesting enough to warrant FPGA acceleration, as most synthesizable tasks do not contain useful computations. The same applies to most other tasks, with the baseline cluster leading to small, single-task clusters representing no significant computation time. The only two exceptions are *texture-syn* and *edgedetect*, which reach the *Goal* regardless of whether transformations are applied. This is expected for *edgedetect*, as it is a fully synthesizable benchmark included for demonstration purposes. As for *texture-syn*, this application uses `structs` in much the same way as the other CortexSuite applications, but keeps pointer arithmetic simple and without any dynamic memory allocations, allowing its synthesis.

Then, we proceed to lift each restriction individually, to gauge their impact on clustering when compared to no lifting. We note the following:

- Lifting dynamic memory allocations does not help increase the size of any cluster, as the benchmarks are severely limited by `struct` pointers.

- Lifting pointer-to-pointer restrictions leads to clusters matching 29.5% and 34.7% of the computation time for *mser* and *stitch*, respectively. Also, in *sift*, while the cluster remains with only a single task, that task is different from the one in the "none lifted" cluster, representing 4.5% of the computation time.

- Lifting `struct` pointer restrictions leads to better results, e.g., *disparity* reaches a cluster representing 76.8% of the computation time. As for *multi-ncut*, lifting this restriction leads to a cluster matching 99.9% of the computation time, and exceeding *Goal*. This application illustrates an extreme, where a single task dominates all computation time.

- Lifting `struct` arguments with pointers leads to no improvements except for *mser*, where we reach a slightly larger cluster representing 5.4% of the computation time.

Table 4.6: Three clusters for offloading obtained for each CortexSuite and *edgedetect* applications, representing the Baseline, Hotspot and Entire Useful regions

Application	**Input Size**	**CPU ET (s)**	**Baseline region (BR)** *Goal* = 80% No restr. lifted		**Hotspot region (HR)** *Goal* = 80% All restr. lifted		**Entire useful region (ER)** *Goal* = 100% All restr. lifted	
			#Task	%comp	#Task	%comp	#Task	%comp
disparity	fullhd	19.60	1	0.0	12	88.6	92	99.2
localization	vga	1.47	1	0.0	486	81.9	1203	99.7
mser	fullhd	3.63	1	0.0	57	94.0	57	94.0
multi-ncut	qcif	51.09	1	0.0	21	99.9	112	100.0
sift	fullhd	18.67	1	0.0	562	98.0	564	98.1
stitch	fullhd	175.03	1	0.0	130	97.6	496	100.0
svm	cif	0.98	1	0.0	2,021	73.6	2,414	98.9
texture-syn	fullhd	110.11	6	73.0	6	73.0	103	98.7
tracking	fullhd	4.56	2	3.4	368	78.6	368	78.6
edgedetect	fullhd	0.31	7	76.0	7	76.0	9	92.0

ET Execution Time, *#Task* Cluster Size, *restr.* HLS restrictions

This analysis shows that each transformation, by itself, is not enough to enable the full cluster of these applications, with $Goal = 80\%$, with two notable exceptions. However, when we enable all transformations, every single application is clustered into a region representing $\approx 80\%$ of the computation time, achieving the specified *Goal*. This shows that the transformations, when applied as a whole, do not cause any destructive interference, and enable other transformations to be more successful (i.e., positive interference). Therefore, we see no reason to restrict the usage of any transformation, and can safely apply the cluster algorithm assuming that every non-synthesizable task can be transformed.

In Table 4.6, we summarize three clusters and their underlying regions: the Baseline Region (BR), the Hotspot Region (HR), and the Entire Useful Region (ER). BR matches the cluster achieved in Table 4.5 with $Goal = 80\%$ and without lifting any HLS restriction, reflecting the largest possible region we could offload without applying our method. This provides us with a lower bound for partitioning. HR matches the cluster achieved in Table 4.5 when lifting all HLS restrictions, and again with $Goal = 80\%$. Finally, ER matches a cluster achieved using $Goal = 100\%$, and by lifting all HLS restrictions. This represents an upper-bound for partitioning, since each cluster consists of the entire ETG.

To finalize, we also consider the impact that the transformations have on the complexity of the final source code. In Tables 4.7 and 4.8, we show the Frama-C [156] statistics for three regions matching three clusters: the baseline region (BR), where no restrictions are lifted, and the hotspot (HR) and entire useful (ER) regions. We show the results before and after applying the transformations, as well as the change, in percentage, of each metric after transforming the code.

We start with Table 4.7, which focuses on statement-level metrics (Statement Lines of Code (SLOC), number of assignments, and number of pointer dereferences). SLOC count exposes the simplicity of the BR regions, which have, for the most part, a single-digit number of statements. This is reflected in the negligible amount of computation time represented by them. For the other

Table 4.7: Frama-C statement-level metrics for the BR, HR and ER code regions pre- and post-transformations

Application	Reg.	Statement Lines of Code		Assignments		Pointer Dereferences	
		Pre	**Post (Inc%)**	**Pre**	**Post (Inc%)**	**Pre**	**Post (Inc%)**
disparity	BR	3	3 (0.0%)	2	2 (0.0%)	4	4 (0.0%)
	HR	139	113 (-18.7%)	61	62 (1.6%)	73	55 (-24.7%)
	ER	511	422 (-17.4%)	197	266 (35.0%)	347	117 (-66.3%)
localization	BR	8	8 (0.0%)	3	3 (0.0%)	12	12 (0.0%)
	HR	2,668	2261 (-15.3%)	1,109	1,514 (36.5%)	2,522	718 (-71.5%)
	ER	6,578	5151 (-21.7%)	2,650	3,466 (30.8%)	6,071	1,546 (-74.5%)
mser	BR	15	15 (0.0%)	6	6 (0.0%)	5	5 (0.0%)
	HR	641	606 (-5.5%)	335	364 (8.7%)	499	151 (-69.7%)
	ER	641	606 (-5.5%)	335	364 (8.7%)	499	151 (-69.7%)
multi-ncut	BR	3	3 (0.0%)	2	2 (0.0%)	4	4 (0.0%)
	HR	142	97 (-31.7%)	59	55 (-6.8%)	143	26 (-81.8%)
	ER	638	485 (-24.0%)	263	309 (17.5%)	630	193 (-69.4%)
sift	BR	2	2 (0.0%)	1	1 (0.0%)	1	1 (0.0%)
	HR	3,567	2956 (-17.1%)	1,497	1,852 (23.7%)	2,981	791 (-73.5%)
	ER	3,607	2992 (-17.1%)	1,515	1,870 (23.4%)	2,991	801 (-73.2%)
stitch	BR	5	5 (0.0%)	4	4 (0.0%)	3	3 (0.0%)
	HR	634	491 (-22.6%)	249	329 (32.1%)	647	175 (-73.0%)
	ER	2,759	2280 (-17.4%)	1,162	1,516 (30.5%)	2,464	647 (-73.7%)
svm	BR	6	6 (0.0%)	5	5 (0.0%)	5	5 (0.0%)
	HR	9,897	6887 (-30.4%)	3,950	3,910 (-1.0%)	8,430	3,161 (-62.5%)
	ER	11,654	8049 (-30.9%)	4,564	4,603 (0.9%)	9,952	3,677 (-63.1%)
texture-syn	BR	75	75 (0.0%)	43	43 (0.0%)	75	75 (0.0%)
	HR	91	75 (-17.6%)	49	43 (-12.2%)	65	75 (15.4%)
	ER	1,000	783 (-21.7%)	471	457 (-3.0%)	606	353 (-41.7%)
tracking	BR	34	34 (0.0%)	25	25 (0.0%)	52	52 (0.0%)
	HR	2,127	1834 (-13.8%)	932	1,218 (30.7%)	2,043	490 (-76.0%)
	ER	2,127	1834 (-13.8%)	932	1,218 (30.7%)	2,043	490 (-76.0%)
edgedetect	BR	183	183 (0.0%)	94	94 (0.0%)	34	34 (0.0%)
	HR	192	183 (-4.7%)	91	94 (3.3%)	34	34 (0.0%)
	ER	227	215 (-5.3%)	112	115 (2.7%)	47	47 (0.0%)

Reg. region, *Pre* Pre-transformations, *Post* Post-transformations, *Inc%* Percentage of increase vs. pre-transformations

regions, SLOC always decreases after applying the transformations. This is due to inlining, which removes function calls, and the elimination of unreachable code during that process. Pointer dereferences also decrease significantly across the board after transformations. This is due to both the inliner simplifying the arithmetic, and to struct flattening, which converts previous pointer accesses to scalar fields into scalar variables. On the other hand, the number of assignments increases for most applications. This is due to extensive variable aliasing caused by the inliner, hindering readability.

Table 4.8 focuses on control-flow structure metrics, such as the number of If/else constructs, number of loops, number of `gotos`, and a McCabe's Cyclomatic Complexity (CC) metric (i.e., a measure of the number of decision points in the program's CFG) [157]. BR regions, once again,

Table 4.8: Frama-C control flow-level metrics for the BR, HR and ER code regions pre- and post-transformations

Application	Reg.	Ifs		Loops		`gotos`		Cyclomatic Complexity	
		Pre	**Post (Inc%)**	**Pre**	**Post (Inc%)**	**Pre**	**Post (Inc%)**	**Pre**	**Post (Inc%)**
disparity	BR	0	0 (0.0%)	0	0 (0.0%)	0	0 (0.0%)	1	1 (0.0%)
	HR	16	14 (-12.5%)	14	12 (-14.3%)	0	0 (0.0%)	26	15 (-42.3%)
	ER	49	47 (-4.1%)	37	35 (-5.4%)	0	0 (0.0%)	122	48 (-60.7%)
localization	BR	1	1 (0.0%)	1	1 (0.0%)	0	0 (0.0%)	2	2 (0.0%)
	HR	224	220 (-1.8%)	153	149 (-2.6%)	1	1 (0.0%)	611	221 (-63.8%)
	ER	594	522 (-12.1%)	397	328 (-17.4%)	4	4 (0.0%)	1,547	523 (-66.2%)
mser	BR	2	2 (0.0%)	1	1 (0.0%)	1	1 (0.0%)	3	3 (0.0%)
	HR	86	86 (0.0%)	34	34 (0.0%)	9	9 (0.0%)	115	87 (-24.3%)
	ER	86	86 (0.0%)	34	34 (0.0%)	9	9 (0.0%)	115	87 (-24.3%)
multi-ncut	BR	0	0 (0.0%)	0	0 (0.0%)	0	0 (0.0%)	1	1 (0.0%)
	HR	16	12 (-25.0%)	10	7 (-30.0%)	1	1 (0.0%)	33	13 (-60.6%)
	ER	66	62 (-6.1%)	35	32 (-8.6%)	1	1 (0.0%)	156	63 (-59.6%)
sift	BR	0	0 (0.0%)	0	0 (0.0%)	0	0 (0.0%)	1	1 (0.0%)
	HR	386	338 (-12.4%)	176	139 (-21.0%)	62	58 (-6.5%)	827	405 (-51.0%)
	ER	392	344 (-12.2%)	180	143 (-20.6%)	62	58 (-6.5%)	835	411 (-50.8%)
stitch	BR	0	0 (0.0%)	0	0 (0.0%)	0	0 (0.0%)	1	1 (0.0%)
	HR	62	58 (-6.5%)	35	32 (-8.6%)	1	1 (0.0%)	162	59 (-63.6%)
	ER	287	280 (-2.4%)	139	133 (-4.3%)	3	3 (0.0%)	674	281 (-58.3%)
svm	BR	0	0 (0.0%)	0	0 (0.0%)	0	0 (0.0%)	1	1 (0.0%)
	HR	1,029	804 (-21.9%)	353	134 (-62.0%)	122	120 (-1.6%)	2,640	1,437 (-45.6%)
	ER	1,185	914 (-22.9%)	457	192 (-58.0%)	123	121 (-1.6%)	3,115	1,626 (-47.8%)
texture-syn	BR	10	10 (0.0%)	7	7 (0.0%)	0	0 (0.0%)	11	11 (0.0%)
	HR	10	10 (0.0%)	7	7 (0.0%)	0	0 (0.0%)	16	11 (-31.3%)
	ER	121	114 (-5.8%)	33	30 (-9.1%)	40	39 (-2.5%)	222	127 (-42.8%)
tracking	BR	2	2 (0.0%)	2	2 (0.0%)	0	0 (0.0%)	3	3 (0.0%)
	HR	190	190 (0.0%)	113	113 (0.0%)	16	16 (0.0%)	479	191 (-60.1%)
	ER	190	190 (0.0%)	113	113 (0.0%)	16	16 (0.0%)	479	191 (-60.1%)
edgedetect	BR	25	25 (0.0%)	20	20 (0.0%)	0	0 (0.0%)	26	26 (0.0%)
	HR	25	25 (0.0%)	20	20 (0.0%)	0	0 (0.0%)	32	26 (-18.8%)
	ER	27	27 (0.0%)	22	22 (0.0%)	0	0 (0.0%)	36	28 (-22.2%)

Reg. region, *Pre* Pre-transformations, *Post* Post-transformations, *Inc%* Percentage of increase vs. pre-transformations

show extreme simplicity. As for the other regions, we note that the number of If/else and loop constructs decreases, once again in part due to the elimination of unreachable code during inlining. The number of `gotos` generally stays the same, although we also notice some reductions, caused by inlining. Finally, CC is quite high for some benchmarks, e.g., 2,640 for *svm* HR. This may signify a heavier burden on HLS, as higher CC values can be reflective of complex CFGs that complicate pipelining.

In summary, we have several takeaways from this final experiment. Firstly, we showed how the cluster algorithm is limited by non-synthesizable tasks, which lead to clusters of little to no utility representing a negligible amount of computation time. By enabling all code transformations lifting HLS restrictions, the algorithm is capable of creating clusters matching the target of $\approx 80\%$ of the CPU computation time. Furthermore, the transformations have positively interfered with each other, achieving better results as a whole than when applied individually. Finally, applying transformations to the regions matching the clusters achieved by the algorithm leads to a simplification of the source code.

4.4 Summary

This chapter described all elements necessary for creating a cluster of tasks for offloading onto an FPGA, for an ETG of a generic C application. We started by exposing the fact that, unlike typical HLS-friendly kernels, large C applications often have non-synthesizable C features in code regions suitable for offloading. We proposed several fully automated source-to-source code transformations that tackle synthesizability issues, opening the doors for the clustering of large C applications using a heuristic greedy algorithm. This algorithm aggressively expands several clusters (one per leaf task on the ETG), in order to reach a set goal, such as achieving a cluster that captures $\approx 80\%$ of the original program's CPU computation time. Given this suite of transformations and the algorithm, we proceeded to first measure the impact some of these transformations have on HLS (e.g., the impact of asynchronous host calls and hoisting memory allocations), which proved to be negligible in terms of both latency and resources. Finally, we applied the clustering algorithm to all applications from CortexSuite, plus *edgedetect*. Results show that without lifting synthesizability constraints, the algorithm can seldom create clusters representing a meaningful percentage of computation. However, by allowing the lifting of all restrictions through transformations, the algorithm creates clusters capturing the goal of $\approx 80\%$ of computation time.

Chapter 5

Optimizing Clusters

In this chapter, we complete the holistic Hardware/Software partitioning and optimization. We start by detailing our method to optimize the region underlying a cluster obtained through the clustering algorithm, focusing on mapping data from off-chip shared memory onto on-chip memory. This optimization consists of three heuristics that use holistic information from the entire program beyond the region to decide whether it can and should map a communicated buffer to on-chip memory. Then, we proceed to the complete validation of the entire process. We start by synthesizing the regions obtained by the clustering algorithm in the previous chapter, and study their potential and limitations compared to their CPU equivalents. Next, we conduct an experiment concerning the different configurations of heuristics for optimizing the regions, which is followed by synthesis and another comparison against the CPU baseline. We conclude the chapter by providing some additional insights into how some regions can be further improved through additional holistic optimizations.

5.1 Holistic On-chip Memory Mapping

The transformations we described enable code region expansions for FPGA offloading, but do not optimize those regions for HLS, and guiding the HW/SW partitioning by the largest regions may not be the best decision. For instance, larger regions may involve more communication costs. On a typical system with a CPU and an FPGA, utilizing shared DRAM, communication costs are incurred every time there is access to data on DRAM, rather than on the FPGA's on-chip memory, e.g., BRAMs and URAMs. With this in mind, we propose heuristics to reduce the communication cost by maximizing on-chip memory usage. Our algorithm focuses on BRAMs only, but could be extended to URAMs by trying to maximize BRAM usage first, and using URAM as a backup due to their higher latency.

We show this algorithm in Algorithm 2. The first step is to synthesize the cluster's region to obtain the initial BRAM usage. Then, we use the ETG to determine the *in_buffer* and *out_buffer* sets. The former represents data buffers that are allocated and initialized before the region. By tracing the paths of the data buffers on the cluster's in-edges up the ETG hierarchy, the algorithm

Algorithm 2 Heuristic BRAM Mapping

```
1:  procedure ApplyOnChipMapping(ETG, Cluster, Recipe, TargetProperties)
2:      BRAM_usage ← Synthesize(Cluster)
3:      BRAM_max ← BRAM × 0.9, BRAM ∈ TargetProperties
4:      all_buffers ← CalculateData(ETG, Cluster)
5:      (in_buffers, out_buffers) ← CalculateInOuts(ETG, Cluster)
6:
7:      exclusive_in_buffers ← in_buffers \ out_buffers
8:      exclusive_in_buffers ← SortByAscendingSize(exclusive_in_buffers)
9:      local_buffers ← all_buffers \ (in_buffers ∪ out_buffers)
10:     local_buffers ← SortByAscendingSize(local_buffers)
11:     if ∃i ∈ {1,...,n} : X_i = A then
12:         HeuristicA(BRAM_usage, BRAM_max, local_buffers)
13:     end if
14:     if ∃i ∈ {1,...,n} : X_i = B then
15:         HeuristicB(BRAM_usage, BRAM_max, exclusive_in_buffers)
16:     end if
17:     if ∃i ∈ {1,...,n} : X_i = C then
18:         HeuristicC(BRAM_usage, BRAM_max, exclusive_in_buffers)
19:     end if
20: end procedure
```

determines if the buffer has been allocated and written to at least once, and if that allocation holds until it reaches the cluster. Conversely, in-edges with no allocation or no data are excluded from this set.

A similar approach applies to the *out_buffer* set, as it contains data buffers with initialized data that are used by the host after the region finishes executing. Data validity is ensured by examining the buffer's data path inside the cluster, to determine whether they contain valid data at all exit points, and the data path outside the cluster and into the host tasks, to determine whether the data is used or ignored.

All buffers that do not map to at least one set are considered to be *local_buffers*, that is, allocated buffers whose entire lifetime exists entirely within the cluster, despite being allocated on the

Algorithm 3 Heuristic A: fully map local buffers

```
1: procedure HeuristicA(BRAM_usage, BRAM_max, local_buffers)
2:     for all buffer b in local_buffers do
3:         if BRAM_usage + b.size < BRAM_max then
4:             MapToBRAM(b)
5:             BRAM_usage ← BRAM_usage + b.size
6:         end if
7:     end for
8: end procedure
```

Algorithm 4 Heuristic B: fully map in-buffers

```
1:  procedure HEURISTICB(BRAM_usage, BRAM_max, exclusive_in_buffers)
2:      if BRAM_usage < BRAM_max then
3:          for all buffer b in exclusive_in_buffers do
4:              if b.size < threshold then
5:                  MAPTOREGISTERS(b)
6:                  ADDMEMCPY(b)
7:                  exclusive_in_buffers ← exclusive_in_buffers \ {b}
8:              else
9:                  if BRAM_usage + b.size < BRAM_max then
10:                     MAPTOBRAM(b)
11:                     BRAM_usage ← BRAM_usage + b.size
12:                     exclusive_in_buffers ← exclusive_in_buffers \ {b}
13:                     ADDMEMCPY(b)
14:                 end if
15:             end if
16:         end for
17:     end if
18: end procedure
```

host. Most of these buffers come as a result of hoisting `malloc()` calls. Having categorized data buffers across these sets, the algorithm applies up to three BRAM mapping heuristics $\langle A, B, C \rangle$. Some or all heuristics may be omitted, depending on a configurable recipe, but their order is always enforced. For example, $\langle A, B \rangle$ and $\langle A, C \rangle$ are valid recipes, but not $\langle C, A \rangle$ or $\langle B, A \rangle$. This is due to the increased specificity of these heuristics, with A being the most generic, and C the most specific.

Heuristic A, as shown in Algorithm 3, focuses on *local_buffers*, and attempts to map as many of them to BRAM as possible. As these buffers are not needed by the host, either before or after the region executes, they stand to gain the most by residing entirely within on-chip memory, as they leverage no advantages of off-chip memory beyond larger storage. The heuristic sorts each local buffer by size, mapping them to on-chip memory. Buffers of size smaller than a given threshold (32 bytes, by default) are mapped to registers, avoiding any BRAM costs. Buffers larger than the threshold are mapped to BRAM, and their cost is added to the total memory mapped on-chip. We note that this value is an estimate: *BRAM_size* is the total size of BRAMs available on-chip, and *BRAM_usage* is the total size of the buffers mapped. In reality, as BRAMs have a fixed size (e.g., 16KB), and as smaller buffers may be each allocated to individual BRAMs, some BRAMs may be underutilized in the final design. To account for this, we stop the algorithm if *BRAM_usage* reaches 90% of *BRAM_size*.

Heuristic B is shown in Algorithm 4, and focuses on *exclusive_in_buffers*, i.e., input buffers that are not used by the host after the region finishes executing. The heuristic attempts to fully map them to BRAMs, from smallest to largest, using the same threshold criterion as Heuristic A. As they hold live data going into the region, each buffer needs to be fully copied from off-chip to on-chip memory using `memcpy()`. This represents the only off-chip memory accesses for each

Algorithm 5 Heuristic C: partially map most promising in-buffer

```
1:  procedure HEURISTICC(BRAM_usage, BRAM_max, exclusive_in_buffers)
2:      if BRAM_usage < BRAM_max then
3:          BRAM_remaining ← BRAM_max − BRAM_usage
4:          b* ← MAXNUMBEROFREADS(exclusive_in_buffers)
5:          factor ← ⌈ b*.length / BRAM_remaining ⌉
6:          if factor ≤ 4 then
7:              chunk_size ← MAPCHUNKTOBRAM(b*, factor)
8:              BRAM_usage ← BRAM_usage + chunk_size
9:              ADDMEMCPY(b*, factor) return
10:         end if
11:     end if
12: end procedure
```

buffer, as the on-chip allocation does not need to be copied back to DRAM. Both this and the previous heuristic may not be exhaustive, as the algorithm may reach *BRAM_max* before every buffer is mapped, or there may be buffers that are too big to map on the available space.

Heuristic C, depicted in Algorithm 5, focuses on an edge case, where there is free memory after applying the previous two heuristics, and *exclusive_in_buffers* that are too big to map fully to BRAMs. In this scenario, the heuristic maps only a chunk of the buffer to on-chip memory, and updates every memory access to either use on-chip or off-chip memory depending on the memory position. This is applied only once, over the most promising remaining buffer in *exclusive_in_buffers*, which is the buffer with the most read/write operations performed on the region. The ETG provides this information, as it is annotated with the maximum number of iterations of each loop, as obtained through profiling, allowing for the static computation of read-/write counts. An additional constraint ensures that the size of the on-chip chunk is regulated by $factor \in \{2,3,4\}$, that is, the chunk can represent either 50%, $\approx$ 33%, or 25% of the original buffer. This factor is determined by the size of the buffer and the on-chip memory remaining. This prevents the creation of very small chunks where most data is in off-chip memory, while allowing for the decision between loading from on-chip and off-chip memory to be defined by a simple modulo operation over the index. Finally, data is mapped to the chunk cyclically, e.g., a buffer $A = [1,2,3,4,5,6,...]$ mapped with $factor = 3$ would have an on-chip chunk $S = [1,4,...]$, governed by memory accesses `idx % 3 ? A[idx] : S[idx]`.

We show an example of all three heuristics in Figure 5.1. We start with the region annotated with information about the initial BRAM usage of 1,892 bytes, as obtained through Vitis HLS, and liveness and size information, as obtained through profiling (i.e., ETG annotations represented as directives in the example). Buffers `A` and `B` have no data initialized entering the cluster, and are not used by the host afterward. Therefore, Heuristic A maps both to local buffers, removing them from the function signature. As the size of `A` is under the 32 bytes threshold, it is mapped as registers (i.e., the compiler does not introduce any explicit BRAM mapping directive). Buffers `C` and `D` have

```
// Region before heuristic BRAM mapping
void region(char *A, char *B, char *C, char *D, char *E, char *F) {
#pragma clava initial_bram_usage = 4 total_bram_count = 1824 bytes_per_bram = 2048
// i.e., BRAM;_usage = 8192 bytes, BRAM_max = 1824 * 2048 * 0.9 = 3361996 bytes
#pragma clava param = A in = NONE out = NONE size = 4
#pragma clava param = B in = NONE out = NONE size = 1024
#pragma clava param = C in = LIVE out = NONE size = 220000
#pragma clava param = D in = LIVE out = NONE size = 1140000
#pragma clava param = E in = LIVE out = NONE size = 3980000
#pragma clava param = F in = LIVE out = LIVE size = 1024
    // region body
}

// Region after heuristic BRAM mapping
void region(char *C, char *D, char *E, char *F) {
    // *A mapped by Heuristic A into registers
    char A[4];

    // *B mapped by Heuristic A into BRAMs
    char B[1024];

    // *C mapped by Heuristic B into BRAMs
    char C_onchip[220000];
    memcpy(C_onchip, C, 220000);

    // *D mapped by Heuristic B into BRAMs
    char D_onchip[1140000];
    memcpy(D_onchip, D, 1140000);

    // *E mapped by Heuristic C into a BRAM slice of factor 2
    char E_onchip_slice[1990000];
    for (int i = 0; i < 1990000; i++) {
        E_onchip_slice[i] = E[i * 2]; // cyclic mapping
    }
#pragma HLS bind_storage impl = BRAM variable = B
#pragma HLS bind_storage impl = BRAM variable = C_onchip
#pragma HLS bind_storage impl = BRAM variable = D_onchip
#pragma HLS bind_storage impl = BRAM variable = E_onchip_slice
    // region body
}
```

Figure 5.1: Example of a code region before and after applying the heuristic BRAM mapping algorithm

initialized data coming into the region, but are not used afterward. They are valid candidates for Heuristic B, which maps them completely to on-chip memory, as there is still enough space. At this point, the BRAM usage estimation is $BRAM\_usage = 8,192 + 1,024 + 220,000 + 1,140,000 = 1,369,216$ bytes (40.7%). Buffer `F` could be another candidate, as it fits on the available space, but as its data is used by the host afterward, it is not considered. The only remaining buffer is `E`, whose size exceeds the availability of BRAMs. It is the only candidate for Heuristic C, and the smallest factor of 2 is enough to create a chunk that fits on the available space. Having mapped half of this buffer to BRAM, the final usage is 3,359,216 bytes (99.9%).

5.2 Final Evaluation

In this section, we provide the final evaluation of our holistic approach toward partitioning and optimizing large applications for FPGAs. We start by synthesizing the clusters obtained by the

Table 5.1: HLS resource usage post Place-and-Route, for the BR, HR, and ER code regions of each application

Application	Reg.	Invocations	FF (%)	LUT (%)	BRAM (%)	DSP (%)	P&R Time (hh:mm:ss)
disparity	BR	64	2682 (0.49%)	1993 (0.73%)	2 (0.11%)	0 (0.0%)	00:08:07
	HR	64	11839 (2.16%)	9470 (3.46%)	4 (0.22%)	43 (1.71%)	00:14:26
	ER	1	23053 (4.21%)	18741 (6.84%)	4 (0.22%)	84 (3.33%)	00:19:31
localization	BR	1	3934 (0.72%)	3071 (1.12%)	2 (0.11%)	5 (0.2%)	00:09:24
	HR	1	108242 (19.75%)	83253 (30.38%)	8 (0.44%)	187 (7.42%)	02:09:59
	ER	1	–	–	–	–	OOM
mser	BR	10,368,000	2560 (0.47%)	2083 (0.76%)	2 (0.11%)	0 (0.0%)	00:08:42
	HR	1	35847 (6.54%)	31400 (11.46%)	6 (0.33%)	9 (0.36%)	00:25:21
	ER	1	35847 (6.54%)	31399 (11.46%)	6 (0.33%)	9 (0.36%)	00:26:32
multi-ncut	BR	1	2682 (0.49%)	1996 (0.73%)	2 (0.11%)	0 (0.0%)	00:08:35
	HR	1	5534 (1.01%)	4569 (1.67%)	2 (0.11%)	19 (0.75%)	00:09:49
	ER	1	52353 (9.55%)	45410 (16.57%)	184 (10.09%)	43 (1.71%)	00:34:25
sift	BR	1	1867 (0.34%)	1587 (0.58%)	1 (0.05%)	0 (0.0%)	00:08:18
	HR	1	–	–	–	–	–
	ER	1	–	–	–	–	–
stitch	BR	1	1478 (0.27%)	1156 (0.42%)	1 (0.05%)	0 (0.0%)	00:08:04
	HR	1	31510 (5.75%)	24669 (9.0%)	8 (0.44%)	26 (1.03%)	00:27:27
	ER	1	181998 (33.2%)	135746 (49.53%)	31 (1.7%)	254 (10.08%)	02:35:07
svm	BR	1	1733 (0.32%)	1343 (0.49%)	1 (0.05%)	0 (0.0%)	00:08:15
	HR	1	–	–	–	–	OOM
	ER	1	–	–	–	–	OOM
texture-syn	BR	914,880	12738 (2.32%)	10358 (3.78%)	4 (0.22%)	19 (0.75%)	00:13:40
	HR	914,880	12738 (2.32%)	10355 (3.78%)	4 (0.22%)	19 (0.75%)	00:13:38
	ER	1	34264 (6.25%)	29709 (10.84%)	4 (0.22%)	85 (3.37%)	00:42:01
tracking	BR	800	9806 (1.79%)	8192 (2.99%)	2 (0.11%)	51 (2.02%)	00:14:02
	HR	4	141784 (25.87%)	102187 (37.28%)	16 (0.88%)	278 (11.03%)	01:40:45
	ER	4	140564 (25.64%)	102150 (37.27%)	16 (0.88%)	278 (11.03%)	01:43:32
edgedetect	BR	1	46293 (8.45%)	19247 (7.02%)	31 (1.7%)	18 (0.71%)	0:12:14
	HR	1	46293 (8.45%)	19233 (7.02%)	31 (1.7%)	18 (0.71%)	0:12:21
	ER	1	51612 (9.42%)	22173 (8.09%)	31 (1.7%)	30 (1.19%)	0:14:02

Reg. Region, *BR* Baseline Region, *HR* Hotspot Region, *ER* Entire Useful Region, *OOM* Out-of-Memory

clustering algorithm, without applying transformations, in order to measure their suitability compared to the original CPU baseline. Then, we study the impact of the heuristic on-chip memory mapping algorithm using different recipes. We subsequently choose the best recipes for each application, apply the mapping transformations, and synthesize the results. This forms a complete partitioning and optimization flow. We compare these results to their unoptimized versions, and reflect on the strengths of our approach, as well as its limitations. Finally, we provide a few more handcrafted transformations informed by a holistic view of the program that may help overcome some of the identified limitations.

5.2.1 Synthesizing Clusters Prior to Optimization

Having decided on potential regions for offloading, we synthesize them using Vitis HLS, up to place-and-route (P&R). We use the HLS directives automatically selected and configured by Vitis HLS, and run P&R using the default settings with a target clock period of 6.67 ns (150 MHz). We show the resource usage achieved after P&R in Table 5.1. First, we note that some of these regions are executed multiple times during the application's runtime. We provide the worst-case latency

estimate, obtained by annotating loops with their maximum number of iterations (as all data is dynamic, we use values that match the same inputs used for CPU profiling). Execution time is obtained by combining the worst-case latency estimation with the maximum frequency achieved post P&R. This gives us the upper bound, in seconds, of the time it takes to execute these regions on the board. This means that, in a real-world scenario, execution times would likely be shorter, but never longer. We also provide the resources utilized by the final bitstream, and the time it took to run P&R on a workstation with an Intel Xeon E5-2630 @2.40 GHz and 128 GB of RAM.

We start by noting that the *sift* application fails to synthesize to HDL, not even reaching P&R. This is due to the existence of arrays of structs that become apparent after applying the synthesizability transformations, and in particular the region inlining transformation. Our struct flattening transformation explicitly does not support the flattening of arrays of structs (in itself part of the larger problem of converting arrays of structs into structs of arrays, and vice versa). As this scenario is out of scope, we drop the *sift* benchmark from our evaluation going forward.

Table 5.2: Latency, maximum frequency, and execution time achieved post Place-and-Route, for the BR, HR, and ER code regions of each application

Application	Reg.	Invocations	Average Estimated Latency (cycles)	Average Estimated Exec. Time (s)	Worst Estimated Latency (cycles)	Worst Estimated Exec. Time (s)	MaxFreq (MHz)
disparity	BR	64	282	0.0	282	0.0	371.61
	HR	64	163,380,252	54.6	648,764,764	216.7	191.64
	ER	1	3,823,153,966	0.0	61,173,755,938	359.0	170.42
localization	BR	1	6,259	0.0	12,293	0.0	300.21
	HR	1	–	–	1,178,236,093	7.6	155.01
	ER	1	–	–	1,178,236,166	5.5	*214.27
mser	BR	10,368,000	433	14.4	649	21.6	312.21
	HR	1	–	–	102,797,696,795	646.0	159.13
	ER	1	–	–	102,797,696,795	613.0	167.70
multi-ncut	BR	1	282	0.0	282	0.0	356.63
	HR	1	–	–	16,944,193,564	70.4	240.67
	ER	1	–	–	17,237,278,281	111.1	155.16
stitch	BR	1	72	0.0	72	0.0	414.08
	HR	1	205,321,964,725	1223.1	1,868,095,891,828	11128.2	167.87
	ER	1	210,567,792,254	0.0	1,907,661,718,711	12683.9	150.4
svm	BR	1	74	0.0	74	0.0	280.66
	HR	1	–	–	990,445,268,455	5368.8	*184.48
	ER	1	–	–	990,989,749,811	5220.4	*189.83
texture-syn	BR	914,880	149,661	705.4	74,683	352.0	194.10
	HR	914,880	149,661	685.0	694,083	3176.9	199.88
	ER	1	21,536,245,290	137.8	1,632,977,041,549	10451.1	156.25
tracking	BR	800	9,847	0.0	37,531	0.1	227.95
	HR	4	20,648,652	0.5	170,581,504	4.5	153.19
	ER	4	20,648,652	0.5	170,581,504	4.4	153.37
edgedetect	BR	1	115,798,869	0.7	115,798,869	0.7	161.11
	HR	1	115,798,869	0.7	115,798,869	0.7	162.95
	ER	1	122,019,853	0.8	122,019,853	0.8	156.40

Reg. Region, *BR* Baseline Region, *HR* Hotspot Region, *ER* Entire Useful Region. Frequency values with an asterisk (*) are estimates

We further note that the three largest regions, *localization* ER and *svm* HR and ER, fail to finish P&R due to out-of-memory issues, requiring us to use the frequency estimated by Vitis HLS, rather than the one obtained by P&R. While this is a limitation of the underlying workstation, it calls into question the feasibility of synthesizing very large regions. All HR regions (i.e., the regions obtained in order to capture $\approx 80\%$ of the CPU computation time), however, finish P&R

Table 5.3: Application speedups achieved by offloading the unoptimized BR, HR, and ER code regions of each application

Application	Reg.	Lat.	FPGA Region ExecTime (s)	Equiv. CPU Region ExecTime (s)	FPGA Region Speedup vs. CPU	Entire CPU-FPGA application speedup	
						Achieved	Theoretical Upper Bound
disparity	BR	W	0.00	0.00	0.00×	1.00×	1.00×
		A	0.00	0.00	0.00×	1.00×	1.00×
	HR	W	216.66	17.37	0.08×	0.09×	8.77×
		A	54.56	17.37	0.32×	0.35×	8.77×
	ER	W	358.96	19.44	0.05×	0.05×	125.00×
		A	22.43	19.44	0.87×	0.87×	125.00×
localization	BR	W	0.00	0.00	0.00×	1.00×	1.00×
		A	0.00	0.00	0.00×	1.00×	1.00×
	HR	W	7.60	4.26	0.56×	0.61×	5.52×
	ER	W	5.50	5.18	0.94×	0.94×	333.33×
mser	BR	W	21.55	0.00	0.00×	0.31×	1.00×
		A	14.38	0.00	0.00×	0.40×	1.00×
	HR	W	646.00	8.93	0.01×	0.01×	16.67×
	ER	W	612.99	8.93	0.01×	0.02×	16.67×
multi-ncut	BR	W	0.00	0.00	0.00×	1.00×	1.00×
		A	0.00	0.00	0.00×	1.00×	1.00×
	HR	W	70.40	42.76	0.61×	0.61×	1000.00×
	ER	W	111.09	42.80	0.39×	0.39×	∞
stitch	BR	W	0.00	0.00	0.00×	1.00×	1.00×
		A	0.00	0.00	0.00×	1.00×	1.00×
	HR	W	11128.23	354.29	0.03×	0.03×	41.67×
		A	1223.10	354.29	0.29×	0.29×	41.67×
	ER	W	12683.92	363.00	0.03×	0.03×	∞
		A	1400.05	363.00	0.26×	0.26×	∞
svm	BR	W	0.00	0.00	0.00×	1.00×	1.00×
		A	0.00	0.00	0.00×	1.00×	1.00×
	HR	W	5368.85	1.40	0.00×	0.00×	3.79×
	ER	W	5220.41	1.88	0.00×	0.00×	90.91×
texture-syn	BR	W	352.01	124.76	0.35×	0.43×	3.70×
		A	705.42	124.76	0.18×	0.23×	3.70×
	HR	W	3176.92	124.76	0.04×	0.05×	3.70×
		A	685.02	124.76	0.18×	0.23×	3.70×
	ER	W	10451.05	168.68	0.02×	0.02×	76.92×
		A	137.83	168.68	1.22×	1.22×	76.92×
tracking	BR	W	0.13	0.44	3.30×	1.02×	1.04×
		A	0.03	0.44	12.59×	1.03×	1.04×
	HR	W	4.45	10.06	2.26×	1.78×	4.67×
		A	0.54	10.06	18.66×	3.90×	4.67×
	ER	W	4.45	10.06	2.26×	1.78×	4.67×
		A	0.54	10.06	18.68×	3.91×	4.67×
edgedetect	BR	W	0.72	0.24	0.33×	0.39×	4.17×
		A	0.72	0.24	0.33×	0.39×	4.17×
	HR	W	0.71	0.24	0.33×	0.39×	4.17×
		A	0.71	0.24	0.33×	0.39×	4.17×
	ER	W	0.78	0.29	0.37×	0.39×	12.50×
		A	0.78	0.29	0.37×	0.39×	12.50×

Reg. Region, *Lat.* latency, *BR* Baseline Region, *HR* Hotspot Region, *ER* Entire Useful Region, *W* Worst-case latency, *A* Average-case latency

in a practical timeframe. Despite the large size of the clusters, and the modest size of the target FPGA, resource usage remains low and well within the FPGA limits. In particular, BRAM usage does not go above 2% in any scenario. This is expected, as Vitis HLS cannot make decisions about how to map memory regions with unknown (i.e., dynamic) sizes. There is, therefore, potential in refining these regions using the heuristics we propose.

The achieved frequency, latency, and estimated execution time are shown in Table 5.2. We highlight, again, that BR regions have minimal latency, matching the equally minimal percentage of computation covered by their ETG cluster. Some applications, like *mser* and *multi-ncut*, do not have an average-case latency estimation as their control flow relies on data-dependent array indexing. Execution times, by themselves, do not tell the full picture, and need to be compared to the CPU baseline to gauge the speedup. Table 5.3 shows the latency speedups of every region, for both the worst-case latency and the average-case latency, when available. We first compare the speedup of the region against its CPU counterpart. No region is better than the CPU, except for all regions of *tracking*, for both the worst- and average-case latencies. The HR and ER regions of *tracking* are the same, and the slight difference in speedup (18.66× and 18.68×) is due to the frequency achieved by two different P&R runs. However, the HR and ER regions only represent 78.6% of the CPU computation time, meaning that the remaining 21.4% of the computation time remains the same. This leads to an overall CPU-FPGA application speedup of 3.90× for the average-case latency, and 1.78×for the worst-case latency.

As there are no more significant speedup improvements across the other applications, we conclude that offloading regions without further optimization is not enough to meet our performance target of improving the overall CPU-FPGA execution time. And while, generally, larger regions lead to higher application speedups, the burden placed on HLS starts becoming unmanageable, preventing us from achieving a routed design for three regions. Therefore, we will focus on the HR regions going forward, as they offer the best compromise between the potential for speedup increases after undergoing optimization, while also remaining synthesizable in useful time.

5.2.2 Impact of Heuristics in Holistic Memory Mapping

Having settled on the HR regions as the target for our optimizations, we now focus on how to improve their performance using the heuristic memory mapping techniques. We create different recipes, i.e., combinations of heuristics *A*, *B*, and *C*, and apply each of those recipes to the benchmarks. Tables 5.4 and 5.5 show the three dimensions of this analysis: an overview of the buffers mapped by each heuristic, the estimated BRAM usage when compared to the usage of the actual design post synthesis, and the impact on the worst- and average-case latencies (estimated using a frequency of 150 MHz for all designs). We look at each benchmark individually:

- *disparity*: this application showcases two types of memory mappings: small arrays with sizes below the 32-byte threshold, which are mapped into registers (e.g., 10 buffers mapped by heuristic *B* have an on-chip size of 0), as well as one buffer that is partially mapped by heuristic *C*. The usage estimates are close to the real usage, being +6% higher. As

Table 5.4: BRAM usage and latency across different on-chip memory mapping recipes (*disparity* through *stitch*)

Application	Recipe	Σ Mapped Buffers (on-chip size)	Estimated BRAM Usage %	Actual BRAM Usage %	Average ExecTime (s)	Worst ExecTime (s)
disparity	–	–	0.22	0.44	69.71	276.81
	A	3 (0)	0.22	0.22	68.74	273.07
	B	10 (0)	0.22	0.44	69.55	276.50
	C	1 (2,796,885)	75.09	68.53	195.50	532.27
	A, B	3 (0) + 10 (0)	0.22	0.22	68.64	272.88
	B, C	10 (0) + 1 (2,796,885)	75.09	68.53	195.30	531.87
	A, C	3 (0) + 1 (2,796,885)	75.09	68.31	194.53	528.54
	A, B, C	3 (0) + 10 (0) + 1 (2,796,885)	75.09	68.31	194.39	528.26
localization	–	–	0.44	3.18	–	7.86
	A	146 (1,048,036)	28.49	3.18	–	0.06
	B	11 (8,032)	0.65	3.18	–	7.86
	C	1 (8,000)	0.65	3.18	–	7.86
	A, B	146 (1,048,036) + 11 (8,032)	28.71	3.18	–	0.06
	B, C	11 (8,032) + 0 (0)	0.65	3.18	–	7.86
	A, C	146 (1,048,036) + 1 (8,000)	28.71	3.18	–	0.06
	A, B, C	146 (1,048,036) + 11 (8,032) + 0 (0)	28.71	3.18	–	0.06
mser	–	–	0.33	3.18	–	685.32
	A	15 (30,244)	1.14	30.98	–	493.33
	B	2 (0)	0.33	3.45	–	685.32
	C	1 (2,764,800)	74.34	3.40	–	703.71
	A, B	15 (30,244) + 2 (0)	1.14	31.25	–	493.33
	B, C	2 (0) + 1 (2,764,800)	74.34	3.45	–	703.71
	A, C	15 (30,244) + 1 (2,764,800)	75.15	31.20	–	511.85
	A, B, C	15 (30,244) + 2 (0) + 1 (2,764,800)	75.15	31.25	–	511.85
multi-ncut	–	–	0.11	0.99	–	113.01
	A	6 (736,048)	19.81	0.99	–	112.87
	B	3 (368,024)	9.96	0.99	–	113.01
	C	1 (368,024)	9.96	68.31	–	113.01
	A, B	6 (736,048) + 3 (368,024)	29.67	0.99	–	112.87
	B, C	3 (368024) + 0 (0)	9.96	68.31	–	113.01
	A, C	6 (736,048) + 1 (368,024)	29.67	68.31	–	112.87
	A, B, C	6 (736,048) + 3 (368,024) + 0 (0)	29.67	68.31	–	112.87
stitch	–	–	0.44	0.44	1,368.81	12,453.97
	A	42 (3,114,184)	83.80	18.64	38.75	330.25
	B	3 (667,332)	18.30	9.54	1,368.77	12,453.89
	C	1 (667,332)	18.30	9.54	1,368.77	12,453.89
	A, B	42 (3,114,184) + 2 (0)	83.80	27.74	38.75	330.24
	B, C	3 (667,332) + 0 (0)	18.30	9.54	1,368.77	12,453.89
	A, C	42 (3,114,184) + 1 (333,666)	92.74	27.74	38.75	330.25
	A, B, C	42 (3,114,184) + 2 (0) + 1 (333,666)	92.74	27.74	38.75	330.25

for the worst-case latency, the best result is 272.88 s, achieved by the recipe $\langle A, B \rangle$, being marginally smaller than the baseline latency of 276.81 s. The same holds for the average-case, decreasing from 69.71 s to 68.64 s. Furthermore, all recipes that use heuristic *C* have worse latency than the non-optimized baseline (e.g., $\langle C \rangle$ has an average latency of 195.50 s). Therefore, the heuristics fail to meaningfully improve this application.

- *localization*: as one of the applications most affected by `malloc()` hoisting, it stands to gain the most from heuristic *A*, which maps a total of 146 local buffers onto on-chip memory.

Table 5.5: BRAM usage and latency across different on-chip memory mapping recipes (*svm* through *edgedetect*)

Application	Recipe	Σ Mapped Buffers (on-chip size)	Estimated BRAM Usage %	Actual BRAM Usage %	Average ExecTime (s)	Worst ExecTime (s)
svm	–	–	11.73	0.88	–	4,033.24
	A	214 (200,216)	17.09	58.44	1.44	2,645.95
	B	0 (0)	11.73	17.32	–	4,033.24
	C	0 (0)	11.73	17.32	–	4,033.24
	A, B	214 (200,216) + 0 (0)	17.09	60.86	1.44	2,645.95
	B, C	0 (0) + 0 (0)	11.73	17.32	1.44	2,645.95
	A, C	214 (200,216) + 0 (0)	17.09	66.78	–	4,033.24
	A, B, C	214 (200,216) + 0 (0) + 0 (0)	17.09	69.19	1.44	2,645.95
texture-syn	–	–	0.22	4.82	912.81	4,233.35
	A	0 (0)	0.22	12.06	912.81	4,233.35
	B	17 (9,216)	0.47	4.82	796.23	3,736.98
	C	1 (9,216)	0.47	4.82	847.36	3,933.64
	A, B	0 (0)	0.47	12.06	796.23	3,736.98
	B, C	17 (9,216) + 0 (0)	0.47	12.06	796.23	3,736.98
	A, C	0 (0) + 1 (9,216)	0.47	4.82	847.36	3,933.64
	A, B, C	0 (0) + 17 (9,216) + 0 (0)	0.47	12.06	796.23	3,736.98
tracking	–	–	0.88	0.44	0.14	1.15
	A	104 (2,368,672)	64.29	0.44	0.09	0.75
	B	6 (0)	0.88	3.62	0.14	1.15
	C	0 (0)	0.88	3.62	0.14	1.15
	A, B	104 (2,368,672) + 6 (0)	64.29	3.62	0.09	0.75
	B, C	6 (0) + 0 (0)	0.88	3.62	0.14	1.15
	A, C	104 (2,368,672) + 0 (0)	64.29	3.62	0.09	0.75
	A, B, C	104 (2,368,672) + 6 (0) + 0 (0)	64.29	3.62	0.09	0.75
edgedetect	–	–	1.70	3.40	0.77	0.77
	A	0 (0)	1.70	63.93	0.77	0.77
	B	2 (2,073,600)	57.21	3.40	0.21	0.21
	C	1 (9)	1.70	3.40	0.32	0.32
	A, B	0 (0) + 2 (2,073,600)	57.21	63.93	0.21	0.21
	B, C	2 (2,073,600) + 1 (1,036,800)	84.96	3.40	0.32	0.32
	A, C	0 (0) + 1 (9)	1.70	63.93	2.12	2.12
	A, B, C	0 (0) + 2 (2,073,600) + 1 (1,036,800)	84.96	63.93	2.12	2.12

This contributes to several recipes achieving worst-case latencies of 0.06 s, compared to the baseline of 7.86 s, making their application a success.

- *mser*: this application has several buffers originating from hoisting, but very few buffers of any other type, making heuristic *B* largely ineffective. We achieve the best result by applying only heuristic *A*, leading to a worst-case latency of 493.33 s compared to a baseline of 685.32 s. This application also marks the first case where the BRAM usage estimate is significantly smaller than the actual BRAM usage, for recipes $\langle A \rangle$ and $\langle A, B \rangle$.

- *multi-ncut*: this is a compute-heavy application that uses only 10 buffers. All buffers except for the output buffer are mapped completely by $\langle A, B \rangle$, for a latency of 112.87 s. However, even this optimal mapping is not enough to achieve a meaningful improvement, as the baseline latency is 113.01 s. This is due to the application being governed by a loop where

every iteration has loop-carried dependencies that prevent the HLS tool from extracting any parallelism.

- *stitch*: as another application impacted by aggressive `malloc()` hoisting, it benefits the most from heuristic *A*. Indeed, every recipe that uses *A* achieves an average-case latency reduction from 1,368.1 s to 38.75 s, and a worst-case latency reduction from 12,453.97 s to 330.25 s.

- *svm*: as the largest application in the suite, it is also impacted by hoisted `malloc()` calls, and heuristic *A* is, once again, very effective at mapping 214 buffers. In fact, heuristics *B* and *C* map no buffers at all, regardless of being applied together or separately. Heuristic *A*, by itself, reduces the worst-case latency from 4,033.24 s to 2,645.95 s. It is also worth noting that Vitis is now capable of estimating average-case latencies for all variants where *A* is applied, which implies a reduction in control-flow complexity.

- *texture-synthesis*: as this application, despite having structs, is uniquely synthesizable, it does not contain any hoisted `malloc()` calls, nor any other hoisted calls from other dynamic memory allocations. Furthermore, its small buffers are completely mapped by heuristic *B*, leading to a modest decrease in latency, from 912.81 s to 796.23 s for the average-case, and 4,233.35 s to 3,736.98 s for the worst-case.

- *tracking*: as a medium-sized application with hoisted `malloc()` calls, it benefits from heuristic *A*. Heuristic *B* maps some smaller buffers into registers, and heuristic *C* has no effect. This leads to an average-case latency decrease from 0.14 s to 0.09 s, and a worst-case decrease from 1.15 s to 0.75 s.

- *edgedetect*: finally, this application has only fixed-sized input buffers (and therefore equal average- and worst-case latencies), causing heuristic *A* to be innocuous. The best improvement is achieved by heuristic *B* in isolation, reducing latency from 0.77 s to 0.21 s.

To summarize, we find the recipe $\langle A, B \rangle$ to be the most effective, as well as applying *A* or *B* individually. The best results come from the larger applications, such as *svm* and *localization*, which have high memory footprints due to hundreds of hoisted `malloc()` calls. The smaller applications, like *multi-ncut* and *texture-synthesis*, are harder to optimize using the heuristics, as their memory footprints are small, motivating the need for additional investment into other optimizations complementary to on-chip mapping. Finally, we highlight, in Tables 5.4 and 5.5, the best version of each application, which serve as the basis for the next experiment.

5.2.3 Synthesizing Clusters Post Optimization

In this final experiment, we synthesize and P&R the optimized Hotspot Regions (HR-O) obtained from the previous step. We show the resource usage in Table 5.6, and the latencies in Table 5.7. Finally, we calculate the speedups of the HR-O region compared to its CPU equivalent, as well as

Table 5.6: HLS resource usage post Place-and-Route, for the Optimized Hotspot Region (HR-O) of each application

Application	FF (%)	LUT (%)	BRAM (%)	DSP (%)	P&R Time (hh:mm:ss)
disparity	10325 (1.88%)	8370 (3.05%)	2 (0.11%)	48 (1.90%)	0:08:57
localization	57871 (10.56%)	42768 (15.60%)	513 (28.13%)	239 (9.48%)	0:11:03
mser	36672 (6.69%)	33342 (12.17%)	10 (0.55%)	9 (0.36%)	0:20:16
multi-ncut	4658 (0.85%)	4211 (1.54%)	502 (27.52%)	9 (0.36%)	0:14:36
stitch	9930 (1.81%)	10027 (3.66%)	1108 (60.75%)	27 (1.07%)	0:06:55
svm	26763 (4.88%)	24537 (8.95%)	168 (9.21%)	136 (5.40%)	0:10:01
texture-syn	18935 (3.45%)	11935 (4.35%)	29 (1.59%)	19 (0.75%)	0:14:24
tracking	109681 (20.01%)	80630 (29.42%)	1127 (61.79%)	220 (8.73%)	0:09:26
edgedetect	20833 (3.80%)	17077 (6.23%)	1048 (57.46%)	30 (1.19%)	0:30:32

the speedup of the overall CPU-FPGA application. We show them in Table 5.8. We analyze each application individually:

- *disparity*: this is the only application whose speedup did not improve with optimizations, staying at 0.31×. While latency decreased slightly, there was also a decrease in the maximum frequency which offsets any gains. This is due to the memory optimization heuristics failing to apply almost entirely. The $\langle A, B \rangle$ configuration contributed to the small decrease in latency by mapping very small buffers to registers, but failed to map any input array into BRAM. Most buffers used by *disparity* are too large to fit into BRAM, and heuristic C did not lead to any improvement.

- *localization*: as expected from the results of the optimization, the HR-O region of this application reaches a speedup of 74.82×when compared to its CPU equivalent, for the average-case latency. By aggressively mapping to BRAM buffers coming from hoisted `malloc()` calls, for a final usage of 28.13%, the memory optimizations prove to be a great success. The overall CPU-FPGA application speedup is at 5.21×, which approaches the theoretical upper bound of 5.52×(i.e., the theoretical upper bound is the CPU-FPGA speedup achieved by an FPGA cluster with no latency).

Table 5.7: Application speedups achieved by offloading the optimized Hotspot Region (HR-O) of each application

Application	Invocations	Average Estimated Latency (cycles)	Average Estimated Exec. Time (s)	Worst Estimated Latency (cycles)	Worst Estimated Exec. Time (s)	MaxFreq (MHz)
disparity	64	160,881,617	55.7	639,572,225	221.4	184.88
localization	1	–	–	9,117,020	0.1	160.18
mser	1	–	–	73,999,159,599	379.3	195.08
multi-ncut	1	–	–	16,931,220,443	101.1	167.42
stitch	1	5,812,343,674	36.3	49,536,666,531	309.1	160.28
svm	1	215,385,513	1.3	396,892,653,321	2399.6	165.40
texture-syn	914880	130,546	585.5	612,700	2747.8	204.00
tracking	4	13,513,613	0.4	112,965,649	3.0	153.05
edgedetect	1	31,151,687	0.2	31,151,687	0.2	180.80

Table 5.8: Application speedups achieved by offloading the Hotspot Region, before and after optimization (HR and HR-O)

Application	**Reg.**	**Lat.**	**FPGA Region ExecTime (s)**	**Equiv. CPU Region ExecTime (s)**	**FPGA Region Speedup vs. CPU**	**Entire CPU-FPGA application speedup**	
						Achieved	**Theoretical Upper Bound**
disparity	HR	W	216.66	17.37	0.08×	0.09×	8.77×
		A	54.56	17.37	0.32×	0.35×	
	HR-O	W	221.40	17.37	0.08×	0.09×	
		A	55.69	17.37	0.31×	0.34×	
localization	HR	W	7.60	4.26	0.56×	0.61×	5.52×
	HR-O	W	0.06	4.26	74.82×	5.21×	
mser	HR	W	646.00	8.93	0.01×	0.01×	16.67×
	HR-O	W	379.32	8.93	0.02×	0.03×	
multi-ncut	HR	W	70.40	42.76	0.61×	0.61×	1000.00×
	HR-O	W	101.13	42.76	0.42×	0.42×	
stitch	HR	W	11128.23	354.29	0.03×	0.03×	41.67×
		A	1223.10	354.29	0.29×	0.29×	
	HR-O	W	309.06	354.29	1.15×	1.14×	
		A	36.26	354.29	9.77×	8.07×	
svm	HR	W	5368.85	1.40	0.00×	0.00×	3.79×
	HR-O	W	2399.61	1.40	0.00×	0.00×	
		A	1.30	1.40	1.07×	1.05×	
texture-syn	HR	W	3176.92	124.76	0.04×	0.05×	3.70×
		A	685.02	124.76	0.18×	0.23×	
	HR-O	W	2747.80	124.76	0.05×	0.06×	
		A	585.47	124.76	0.21×	0.27×	
tracking	HR	W	4.45	10.06	2.26×	1.78×	4.67×
		A	0.54	10.06	18.66×	3.90×	
	HR-O	W	2.95	10.06	3.41×	2.25×	
		A	0.35	10.06	28.49×	4.14×	
edgedetect	HR	W	0.71	0.24	0.33×	0.39×	4.17×
		A	0.71	0.24	0.33×	0.39×	
	HR-O	W	0.17	0.24	1.37×	1.26×	
		A	0.17	0.24	1.37×	1.26×	

Reg. Region, *Lat.* latency, *HR* Hotspot Region, *HR-O* Optimized Hotspot Region, *W* Worst-case latency, *A* Average-case latency

- *mser* and *multi-ncut*: we group these together, as they are very similar: both had modest latency improvements, but not enough to accelerate the region. However, this is only reflective of the worst-case latency scenario, as Vitis HLS has once again failed to determine it. It is likely that the average-case scenario leads to speedups, as even *localization*, with a staggering 74.82×speedup for the average-case, has a speedup of only 0.56×for the worst-case scenario. Further work is needed to determine the actual latency of these applications, be it through co-simulation or by running the kernels on the target FPGA.

- *stitch*: as another large application taking good advantage of the memory mapping optimizations, it achieves a speedup of 9.77×when compared to its CPU counterpart. Furthermore, as HR-O accounts for 97.6% of the computation time, the CPU fraction left as-is is only 2.4%, which reflects as an application speedup of 8.07×.

- *svm*: despite only achieving an almost negligible speedup of 1.07×for the equivalent CPU region, and of 1.05×for the entire application, this application was still massively improved by the mapping heuristics. The worst-case latency of the unoptimized HR region was insurmountable, at 9.9×10^{11} cycles, for an execution time of 5,369 s. The optimized version, on the other hand, halves that time to 2,400 s. The optimized version now allows for the determination of an average-case latency of only 1.3 s, which once again shows how spread out the average- and worst-case latencies can be.

- *texture-synthesis*: this application faces a rather unique problem, in that it needs to be invoked a total of 914,880 times, as it is enclosed by the application's main loop. This requires the region to have minimal latency, and while there is a marked decrease from the unoptimized version, from 2,748 s to 585 s, it is still not enough to match the equivalent CPU region. This result may inform a future improvement of the algorithm where we limit, or otherwise penalize, clusters with a high number of invocations.

- *tracking*: similarly to the other large benchmarks, it achieves a speedup of 28.49×compared to the equivalent CPU region, improving on its already useful unoptimized version. The speedup of the entire CPU-FPGA application is 4.14×, approaching the theoretical upper bound of 4.67×.

- *edgedetect*: finally, this application has very large input arrays, which places some limits on memory mapping to BRAMs. Nevertheless, it achieves a modest speedup of 1.37×for the region and 1.26×for the application.

Overall, 5 out of 9 benchmarks achieve useful CPU-FPGA application speedups. As for the remaining 4, we are able to identify their limitations, and suggest ways of further improving them: *disparity* fails because the heuristics are not effective when in the presence of large input data. *mser* and *multi-ncut* require further information to enable Vitis HLS to obtain the average latency, as we presently are limited by the worst-case latency lower bound. Finally, *texture-synthesis* suggests that the clustering algorithm should avoid forming clusters subject to a high number of invocations.

5.3 Summary

In this chapter, we first introduced a set of three on-chip memory mapping heuristics, which aim to improve the latency of the offloaded regions by relying on holistic information from the entire program.

We then proceeded with the evaluation of our methodology: we synthesized all three cluster sizes obtained for each application (BR, HR, and ER), and calculated the speedup of each region compared to its CPU counterpart, and the speedup of the CPU-FPGA application when combining the offloaded region with the remaining host code. The main takeaway is that offloading BR and ER does not lead to satisfactory results, as the former is too small and too fast to cause any impact on the overall latency of the application, and the latter faces insurmountable synthesizability

challenges due to its size, preventing some average-case latencies from being determined, as well as demanding unsustainable amounts of time and memory to finish P&R. We settled, therefore, on the HR regions as the suitable middle ground. Unoptimized HR regions do not have useful speedups for the most part, but can stand to be improved.

Using the HR regions, we applied all combinations of on-chip memory mapping heuristics, and picked the best result for each application. The combination of heuristics *A* and *B* dominated, although some applications favored either *A* or *B*. Heuristic *C*, being an edge-case, was seldom applied, and never to a point of improving synergistically with *A* or *B*.

Finally, we synthesized the optimized HR regions, HR-O, in much the same manner as the original HR regions. We achieved useful speedups for 5 out of 9 applications, with region speedups ranging from 1.07×to 74.82×, and overall CPU-FPGA application speedups ranging from 1.05×to 8.07×. We identified the limitations of the other 4 applications, and discussed some possible solutions.

Chapter 6

Conclusion

In this final chapter, we provide our concluding remarks for the thesis by offering answers for our research questions, followed by a discussion of future research avenues.

6.1 Concluding Remarks

This thesis provided a method leveraging source-to-source compilation to accelerate large C applications on CPU-FPGA systems. We proposed a C subset suitable for analysis, followed by a task graph specification apt for analyzing and transforming an entire application. By using this graph as the input of a greedy clustering algorithm, and by leveraging several code transformations, we can create large clusters of tasks matching code regions covering most of the original CPU computation time. By further optimizing the communication of the offloaded code region by maximizing the usage of on-chip memory, we achieved speedups of the entire CPU-FPGA applications of up to 8.07$\times$ that of the original CPU baseline. We conclude by recalling and answering the research questions driving this thesis:

RQ1: For a large C application, does offloading increasingly large code regions to an FPGA inherently improve the overall CPU-FPGA application latency, or do the communication burden and synthesis constraints eventually limit the acceleration gains?
Our approach shows that larger clusters lead to speedup improvements on the CPU-FPGA applications as a whole, and particularly in the presence of very large applications with an equally large number of tasks, such as *localization* and *stitch*. Indeed, by choosing a reasonable target of capturing $\approx 80\%$ of the original CPU computation time, we achieve clusters that are synthesizable in useful time, without exceeding the available resources. When expanding toward regions encapsulating the entire application, latency only improves slightly for most applications, and even decreases for some, e.g., *multi-ncut* and *stitch*. Synthesis, and in particular Place-and-Routing, also becomes unfeasible for the largest applications. We find, therefore, that an adequate target for the clustering process is paramount towards achieving a cluster that meaningfully improves the application execution time. As for the impact of communication, we find that the least successful benchmarks are the ones that are

computationally intensive, but light on memory accesses, suggesting that the complexity of control-flow and arithmetic intensity should be as much of a limiting factor as communication and synthesis feasibility. Furthermore, the number of times a region is invoked should also be kept low, as shown by *texture-synthesis*.

RQ2: Can source-to-source transformations tackling non-synthesizable C features enable the creation of larger code regions for offloading?

We proposed and validated a collection of source-to-source transformations tackling non-synthesizable C features, from struct flattening to asynchronous calls to the host. Without these transformations, the algorithm cannot find, for most applications, useful regions for offloading, as most tasks are hampered either by non-synthesizable features, or are tightly coupled to other non-synthesizable tasks. This is particularly evident in the CortexSuite applications, where the largest clusters achieved by the algorithm consist of a single task with negligible execution time. Even *llama2*, an application with a synthesizable cluster consisting of 73% of all tasks, sees that number increased to 100% when transforming calls to `printf()` and `fflush()` into asynchronous host calls, allowing us to infer an entire sequence of tokens in a single FPGA execution. Furthermore, the impact of these transformations on synthesis is negligible, and even reduces the overall complexity of the source code. This is of particular import in HLS, as reducing the cyclomatic complexity (and therefore the control-flow complexity) leads to more efficient RTL designs.

RQ3: Do optimizations informed by a holistic understanding of the application lead to competitive FPGA execution times when compared to their CPU counterparts?

By leveraging holistic information of the entire program, including data lifecycles between the host and the FPGA cluster obtained through ETG edges, we categorize communication according to some liveness criteria. These allow for the application of different on-chip mapping heuristics, reducing the burden of communication by reducing accesses to shared DRAM, and by maximizing the usage of BRAMs and registers. These optimizations are paramount in ensuring that the clusters improve the overall execution time of the CPU-FPGA application (e.g., *localization* fails to accelerate the application when unoptimized, with a speedup of $0.61\times$, but achieves a speedup of $5.21\times$ after applying the on-chip mapping heuristics).

RQ4: Can our approach be fully automated?

The entire compilation flow, from the initial reduction to a C subset to the final on-chip memory mapping optimization, is entirely automated. However, this compilation flow still requires considerable information obtained through the CPU profiling of the application, as we require information about both the computation percentage of each task, and the upper bounds for the iterations of dynamic loops and some memory allocations. The burden placed by this restriction upon a hypothetical developer is nevertheless minimal, allowing our approach to be applicable without requiring expert knowledge.

To conclude, let us reflect, for the last time, on the holistic element of our approach. By holistic, we understand that every decision in the process is informed by all elements of the application: we cannot cluster tasks without considering their feasibility, nor can we optimize a region without knowing what lies beyond it. The whole is more than the sum of its parts, and we believe this thesis provides the foundations towards applying this type of methodology not only to applications targeting CPU-FPGA systems, but to any other combinations of applications, accelerators, and performance goals.

6.2 Future Work

We identify several possible paths for future research. Primarily, we find it most relevant to research additional holistic optimizations to further improve the performance of large code regions on FPGAs. Alongside the current on-chip memory mapping heuristics, we intend to exploit intra-cluster transformations, such as task fusion, in order to enable loop fusion and reduce the control-flow complexity of these applications. The on-chip memory mapping heuristics can also be improved further, particularly by exploring criteria beyond buffer size and number of accesses, leveraging queues and streaming instead of copying an entire buffer at once, and creating multiple BRAM copies of the same buffer. Furthermore, we can complement our optimizations using state-of-the-art HLS optimizers, such as ScaleHLS [131]. While it is unclear whether these tools can deal with large (albeit synthesizable) code regions, they can ideally help explore loop-level optimizations with more granularity than our current approach.

The clustering algorithm also requires further research. More specifically, additional refinements during the clustering process need to be considered beyond the aggressive expansion of our current approach. This improved selection process is informed by additional properties on the ETG, and more specifically on its communication edges. Whereas we can currently obtain metrics about the relative parallelism level of a subgraph or the potential for dataflow communication between two tasks, we need more detailed edge annotations based on these properties.

We can also improve the algorithm by expanding our transformations tackling non-synthesizable features. More specifically, we can support more system calls and no-implementation functions if we expand the host call mechanism to support return values, using a synchronous approach instead of an asynchronous one. A synchronous mechanism would require the FPGA kernel to be halted while the call is made, increasing its latency. This cost needs to be balanced with the number of invocations of each synchronous function during partitioning, as well as with the size of the shared memory buffer.

Finally, we recognize the potential that our ETG specification and clustering algorithm have for scenarios beyond FPGA-based Hardware/Software partitioning. As we can create a region of any size for offloading, using any C program as the input, it is feasible to target other accelerators and frameworks with different programming models. We recognize, for example, that there is a straightforward mapping between our task model and OpenMP tasks, making it suitable for a multi-core and GPU context. This can be extended to other task-based frameworks, such as the

OmpSs [158] programming model, capable of targeting multiple different types of accelerators, including FPGAs in its OmpSs@FPGA [159] specialization.

Bibliography

[1] R H Dennard et al. “Design of ion-implanted MOSFET’s with very small physical dimensions”. In: *IEEE Journal of Solid-state Circuits* (1974), p. 13.

[2] H. Esmaeilzadeh et al. “Dark Silicon and the End of Multicore Scaling”. In: *IEEE Micro* 32.3 (May 2012), pp. 122–134. DOI: 10.1109/MM.2012.17.

[3] Nuno Paulino, João Canas Ferreira, and João M. P. Cardoso. “Improving Performance and Energy Consumption in Embedded Systems via Binary Acceleration: A Survey”. In: *ACM Computing Surveys* 53.1 (May 29, 2020), pp. 1–36. DOI: 10.1145/3369764.

[4] Rakhee Kallimani et al. “TinyML: Tools, Applications, Challenges, and Future Research Directions”. In: *Multimedia Tools and Applications* 83.10 (Sept. 9, 2023), pp. 29015–29045. DOI: 10.1007/s11042-023-16740-9. arXiv: 2303.13569[cs].

[5] Russell Tessier, Kenneth Pocek, and Andre DeHon. “Reconfigurable Computing Architectures”. In: *Proceedings of the IEEE* 103.3 (Mar. 2015), pp. 332–354. DOI: 10.1109/JPROC.2014.2386883.

[6] P. Bose. “Computer architecture research: Shifting priorities and newer challenges”. In: *IEEE Micro* 24.6 (Nov. 2004), pp. 5–5. DOI: 10.1109/MM.2004.68.

[7] Abinash Roy, Jingye Xu, and Masud H. Chowdhury. “Multi-core processors: A new way forward and challenges”. In: *2008 International Conference on Microelectronics*. 2008 International Conference on Microelectronics. ISSN: 2159-1679. Dec. 2008, pp. 454–457. DOI: 10.1109/ICM.2008.5393510.

[8] Anindya Banerjee et al. “Navigating the Seismic Shift of Post-Moore Computer Systems Design”. In: *IEEE Micro* 41.6 (Nov. 1, 2021), pp. 162–167. DOI: 10.1109/MM.2021.3114899.

[9] Satnam Singh. “Computing without Processors: Heterogeneous systems allow us to target our programming to the appropriate environment.” In: *Queue* 9.6 (June 2011), pp. 50–63. DOI: 10.1145/1989748.2000516.

[10] Magdy Zahran. “Heterogeneous computing: Here to stay”. In: *Queue* 14 (Nov. 2016), pp. 31–42. DOI: 10.1145/3028687.3038873.

[11] Mark Bohr. "The new era of scaling in an SoC world". In: *2009 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*. 2009 IEEE International Solid-State Circuits Conference - Digest of Technical Papers. ISSN: 2376-8606. Feb. 2009, pp. 23–28. DOI: 10.1109/ISSCC.2009.4977293.

[12] Intel. "Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture". In: (Sept. 2016), p. 482.

[13] Norman P. Jouppi et al. "A domain-specific architecture for deep neural networks". In: *Communications of the ACM* 61.9 (Aug. 22, 2018), pp. 50–59. DOI: 10.1145/3154484.

[14] *Tensor Cores: Versatility for HPC & AI | NVIDIA*. URL: https://www.nvidia.com/en-us/data-center/tensor-cores/ (visited on 07/11/2022).

[15] André Rösti and Michael Franz. *Unlocking the AMD Neural Processing Unit for ML Training on the Client Using Bare-Metal-Programming Tools*. Apr. 3, 2025. DOI: 10.48550/arXiv.2504.03083. arXiv: 2504.03083[cs].

[16] Andrew Boutros and Vaughn Betz. "FPGA Architecture: Principles and Progression". In: *IEEE Circuits and Systems Magazine* 21.2 (2021), pp. 4–29. DOI: 10.1109/MCAS.2021.3071607.

[17] *Xilinx Versal*. URL: https://www.xilinx.com/products/silicon-devices/acap/versal.html (visited on 07/11/2022).

[18] Philippe Coussy et al. "An Introduction to High-Level Synthesis". In: *IEEE Design & Test of Computers* 26.4 (July 2009), pp. 8–17. DOI: 10.1109/MDT.2009.69.

[19] Christian Pilato and Stephanie Soldavini. "Accelerator Design with High-Level Synthesis". In: *Handbook of Computer Architecture*. Ed. by Anupam Chattopadhyay. Singapore: Springer Nature Singapore, 2025, pp. 841–873. DOI: 10.1007/978-981-97-9314-3_19.

[20] Ian Kuon, Russell Tessier, and Jonathan Rose. "FPGA Architecture: Survey and Challenges". In: *Foundations and Trends® in Electronic Design Automation* 2.2 (2008), pp. 135–253. DOI: 10.1561/1000000005.

[21] G. Martin and G. Smith. "High-Level Synthesis: Past, Present, and Future". In: *IEEE Design & Test of Computers* 26.4 (July 2009), pp. 18–25. DOI: 10.1109/MDT.2009.83.

[22] Yi-Hsiang Lai et al. "Programming and Synthesis for Software-defined FPGA Acceleration: Status and Future Prospects". In: *ACM Transactions on Reconfigurable Technology and Systems* 14.4 (Dec. 31, 2021), pp. 1–39. DOI: 10.1145/3469660.

[23] Frank Vahid. "What is Hardware/Software partitioning?" In: *SIGDA Newsl.* 39.6 (June 2009). Number of pages: 1 Place: New York, NY, USA (June 2009), p. 1. DOI: 10.1145/1862900.1862901.

[24] *PolyBench/C – Homepage of Louis-Noël Pouchet*. URL: https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/ (visited on 03/04/2022).

[25] R.K. Gupta and G. De Micheli. “Hardware-software cosynthesis for digital systems”. In: *IEEE Design & Test of Computers* 10.3 (Sept. 1993), pp. 29–41. DOI: 10.1109/54.232470.

[26] F. Vahid and D.D. Gajski. “Specification partitioning for system design”. In: *[1992] Proceedings 29th ACM/IEEE Design Automation Conference*. [1992] 29th ACM/IEEE Design Automation Conference. Anaheim, CA, USA: IEEE Comput. Soc. Press, 1992, pp. 219–224. DOI: 10.1109/DAC.1992.227833.

[27] Jong Kyung Paek, Kiyoung Choi, and Jongeun Lee. “Binary acceleration using coarse-grained reconfigurable architecture”. In: *ACM SIGARCH Computer Architecture News* 38.4 (Sept. 14, 2010), pp. 33–39. DOI: 10.1145/1926367.1926374.

[28] Neng Hou, Xiaohu Yan, and Fazhi He. “A survey on partitioning models, solution algorithms and algorithm parallelization for hardware/software co-design”. In: *Design Automation for Embedded Systems* 23.1 (June 2019), pp. 57–77. DOI: 10.1007/s10617-019-09220-7.

[29] Kizheppatt Vipin and Suhaib A. Fahmy. “FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications”. In: *ACM Comput. Surv.* 51.4 (July 25, 2018), 72:1–72:39. DOI: 10.1145/3193827.

[30] Elena Moscu Panainte, Koen Bertels, and Stamatis Vassiliadis. “The Molen compiler for reconfigurable processors”. In: *ACM Transactions on Embedded Computing Systems* 6.1 (Feb. 2007), p. 6. DOI: 10.1145/1210268.1210274.

[31] Elena Moscu Panainte, Koen Bertels, and Stamatis Vassiliadis. “Interprocedural Compiler Optimization for Partial Run-Time Reconfiguration”. In: *Journal of VLSI signal processing systems for signal, image and video technology* 43.2 (June 2006), pp. 161–172. DOI: 10.1007/s11265-006-7268-0.

[32] R. Niemann and P. Marwedel. “Hardware/software partitioning using integer programming”. In: *Proceedings ED&TC European Design and Test Conference*. ED&TC European Design and Test Conference. ISSN: 1066-1409. Mar. 1996, pp. 473–479. DOI: 10.1109/EDTC.1996.494343.

[33] Mark Bartlett et al. “The Temporal Knapsack Problem and Its Solution”. In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Ed. by Roman Barták and Michela Milano. Red. by David Hutchison et al. Vol. 3524. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 34–48. DOI: 10.1007/11493853_5.

[34] Jigang Wu and Thambipillai Srikanthan. “Low-complex dynamic programming algorithm for hardware/software partitioning”. In: *Information Processing Letters* 98.2 (Apr. 2006), pp. 41–46. DOI: 10.1016/j.ipl.2005.12.008.

[35] Wenjun Shi et al. “Algorithmic Aspects for Bi-Objective Multiple-Choice Hardware/Software Partitioning”. In: *2014 Sixth International Symposium on Parallel Architectures, Algorithms and Programming*. 2014 Sixth International Symposium on Parallel Architectures, Algorithms and Programming (PAAP). Beijing, China: IEEE, July 2014, pp. 7–12. DOI: 10.1109/PAAP.2014.42.

[36] Wu Jigang, Baofang Chang, and Thambipillai Srikanthan. “A Hybrid Branch-and-Bound Strategy for Hardware/Software Partitioning”. In: *2009 Eighth IEEE/ACIS International Conference on Computer and Information Science*. 2009 Eighth IEEE/ACIS International Conference on Computer and Information Science. Shanghai, China: IEEE, 2009, pp. 641–644. DOI: 10.1109/ICIS.2009.152.

[37] Zoltán Ádám Mann, András Orbán, and Péter Arató. “Finding optimal hardware/software partitions”. In: *Formal Methods in System Design* 31.3 (Oct. 30, 2007), pp. 241–263. DOI: 10.1007/s10703-007-0039-0.

[38] Zbigniew Michalewicz and Marc Schoenauer. “Evolutionary Algorithms for Constrained Parameter Optimization Problems”. In: *Evolutionary Computation* 4.1 (Mar. 1996), pp. 1–32. DOI: 10.1162/evco.1996.4.1.1.

[39] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. “Optimization by Simulated Annealing”. In: *Science* 220.4598 (May 13, 1983), pp. 671–680. DOI: 10.1126/science.220.4598.671.

[40] Fred Glover. “Tabu Search—Part II”. In: *ORSA Journal on Computing* 2.1 (Feb. 1990), pp. 4–32. DOI: 10.1287/ijoc.2.1.4.

[41] M. Dorigo, V. Maniezzo, and A. Colorni. “Ant system: optimization by a colony of cooperating agents”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 26.1 (Feb. 1996), pp. 29–41. DOI: 10.1109/3477.484436.

[42] J. Kennedy and R. Eberhart. “Particle swarm optimization”. In: *Proceedings of ICNN’95 - International Conference on Neural Networks*. ICNN’95 - International Conference on Neural Networks. Vol. 4. Perth, WA, Australia: IEEE, 1995, pp. 1942–1948. DOI: 10.1109/ICNN.1995.488968.

[43] B. W. Kernighan and S. Lin. “An Efficient Heuristic Procedure for Partitioning Graphs”. In: *Bell System Technical Journal* 49.2 (Feb. 1970), pp. 291–307. DOI: 10.1002/j.1538-7305.1970.tb01770.x.

[44] T.F. Abdelzaher and K.G. Shin. “Period-based load partitioning and assignment for large real-time applications”. In: *IEEE Transactions on Computers* 49.1 (Jan. 2000), pp. 81–87. DOI: 10.1109/12.822566.

[45] Alok Prakash et al. “Rapid Memory-Aware Selection of Hardware Accelerators in Programmable SoC Design”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26.3 (Mar. 2018), pp. 445–456. DOI: 10.1109/TVLSI.2017.2769125.

[46] J J Hopfield. “Neural networks and physical systems with emergent collective computational abilities.” In: *Proceedings of the National Academy of Sciences* 79.8 (Apr. 1982), pp. 2554–2558. DOI: 10.1073/pnas.79.8.2554.

[47] Jason Cong et al. “FPGA HLS Today: Successes, Challenges, and Opportunities”. In: *ACM Transactions on Reconfigurable Technology and Systems* 15.4 (Dec. 31, 2022), pp. 1–42. DOI: 10.1145/3530775.

[48] Yi-Hsiang Lai et al. “Programming and Synthesis for Software-defined FPGA Acceleration: Status and Future Prospects”. In: *ACM Transactions on Reconfigurable Technology and Systems* 14.4 (Dec. 31, 2021), pp. 1–39. DOI: 10.1145/3469660.

[49] Razvan Nane et al. “A Survey and Evaluation of FPGA High-Level Synthesis Tools”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (Oct. 2016), pp. 1591–1604. DOI: 10.1109/TCAD.2015.2513673.

[50] M. Akif Ozkan et al. “AnyHLS: High-Level Synthesis With Partial Evaluation”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (Nov. 2020), pp. 3202–3214. DOI: 10.1109/TCAD.2020.3012172.

[51] Lan Huang et al. “A Survey on Performance Optimization of High-Level Synthesis Tools”. In: *Journal of Computer Science and Technology* 35.3 (May 2020), pp. 697–720. DOI: 10.1007/s11390-020-9414-8.

[52] *Introduction to Vitis HLS*. URL: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Introduction-to-Vitis-HLS (visited on 07/11/2022).

[53] *GitHub - Xilinx/HLS: Vitis HLS LLVM source code and examples*. URL: https://github.com/Xilinx/HLS (visited on 07/11/2022).

[54] Andrew Canis et al. “From software to accelerators with LegUp high-level synthesis”. In: *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. 2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES). Montreal, QC, Canada: IEEE, Sept. 2013, pp. 1–9. DOI: 10.1109/CASES.2013.6662524.

[55] *Smart High Level Synthesis (HLS) Tool Suite | Microchip Technology*. URL: https://www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/smarthls-compiler (visited on 07/11/2022).

[56] Leandro De Souza Rosa, Christos-Savvas Bouganis, and Vanderlei Bonato. “Non-iterative SDC modulo scheduling for high-level synthesis”. In: *Microprocessors and Microsystems* 86 (Oct. 2021), p. 104334. DOI: 10.1016/j.micpro.2021.104334.

[57] Asif Islam and Nachiket Kapre. “LegUp-NoC: High-Level Synthesis of Loops with Indirect Addressing”. In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). Boulder, CO, USA: IEEE, Apr. 2018, pp. 117–124. DOI: 10.1109/FCCM.2018.00027.

[58] Christian Pilato and Fabrizio Ferrandi. “Bambu: A modular framework for the high level synthesis of memory-intensive applications”. In: *2013 23rd International Conference on Field programmable Logic and Applications*. 2013 23rd International Conference on Field Programmable Logic and Applications (FPL). Porto, Portugal: IEEE, Sept. 2013, pp. 1–4. DOI: 10.1109/FPL.2013.6645550.

[59] Fabrizio Ferrandi et al. “Invited: Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications”. In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 2021 58th ACM/IEEE Design Automation Conference (DAC). San Francisco, CA, USA: IEEE, Dec. 5, 2021, pp. 1327–1330. DOI: 10.1109/DAC18074.2021.9586110.

[60] Lana Josipovic, Andrea Guerrieri, and Paolo Ienne. “From C/C++ Code to High-Performance Dataflow Circuits”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021), pp. 1–1. DOI: 10.1109/TCAD.2021.3105574.

[61] João M. P. Cardoso, Pedro C. Diniz, and Markus Weinhardt. “Compiling for reconfigurable computing: A survey”. In: *ACM Computing Surveys* 42.4 (June 2010), pp. 1–65. DOI: 10.1145/1749603.1749604.

[62] Qijing Huang et al. “The Effect of Compiler Optimizations on High-Level Synthesis-Generated Hardware”. In: *ACM Transactions on Reconfigurable Technology and Systems* 8.3 (May 19, 2015), pp. 1–26. DOI: 10.1145/2629547.

[63] Johannes De Fine Licht et al. “Transformations of High-Level Synthesis Codes for High-Performance Computing”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.5 (May 1, 2021), pp. 1014–1029. DOI: 10.1109/TPDS.2020.3039409.

[64] Yassin Kortli et al. “Hw/Sw Co-Design technique for 2D fast fourier transform algorithm on Zynq SoC”. In: *Integration* 82 (Jan. 2022), pp. 78–88. DOI: 10.1016/j.vlsi.2021.09.005.

[65] Antonios Tragoudaras et al. “Design Space Exploration of a Sparse MobileNetV2 Using High-Level Synthesis and Sparse Matrix Techniques on FPGAs”. In: *Sensors* 22.12 (June 7, 2022), p. 4318. DOI: 10.3390/s22124318.

[66] Prattay Chowdhury and Benjamin Carrion Schafer. “Unlocking Approximations through Selective Source Code Transformations”. In: *Proceedings of the 2021 on Great Lakes Symposium on VLSI*. GLSVLSI ’21: Great Lakes Symposium on VLSI 2021. Virtual Event USA: ACM, June 22, 2021, pp. 359–364. DOI: 10.1145/3453688.3461498.

[67] Jason Cong et al. “Latte: Locality Aware Transformation for High-Level Synthesis”. In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). Boulder, CO: IEEE, Apr. 2018, pp. 125–128. DOI: 10.1109/FCCM.2018.00028.

[68] W. Ye et al. “Fast timing analysis for hardware-software co-synthesis”. In: *Proceedings of 1993 IEEE International Conference on Computer Design ICCD’93*. 1993 IEEE International Conference on Computer Design ICCD’93. Cambridge, MA, USA: IEEE Comput. Soc. Press, 1993, pp. 452–457. DOI: 10.1109/ICCD.1993.393335.

[69] Louis-Noel Pouchet et al. “Polyhedral-based data reuse optimization for configurable computing”. In: *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays - FPGA ’13*. the ACM/SIGDA international symposium. Monterey, California, USA: ACM Press, 2013, p. 29. DOI: 10.1145/2435264.2435273.

[70] Jianyi Cheng et al. “Efficient Memory Arbitration in High-Level Synthesis From Multi-Threaded Code”. In: *IEEE Transactions on Computers* 71.4 (Apr. 1, 2022), pp. 933–946. DOI: 10.1109/TC.2021.3066466.

[71] Julian Oppermann et al. “ILP-based modulo scheduling for high-level synthesis”. In: *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems - CASES ’16*. the International Conference. Pittsburgh, Pennsylvania: ACM Press, 2016, pp. 1–10. DOI: 10.1145/2968455.2968512.

[72] Sonia Mami, Younes Lahbib, and Abdelkader Mami. “A New HLS Allocation Algorithm for Efficient DSP Utilization in FPGAs”. In: *Journal of Signal Processing Systems* 92.2 (Feb. 2020), pp. 153–171. DOI: 10.1007/s11265-019-01454-9.

[73] Zi Wang and Benjamin Carrion Schafer. “Learning from the Past: Efficient High-level Synthesis Design Space Exploration for FPGAs”. In: *ACM Transactions on Design Automation of Electronic Systems* 27.4 (July 31, 2022), pp. 1–23. DOI: 10.1145/3495531.

[74] Vasileios Tsoutsouras et al. “An Exploration Framework for Efficient High-Level Synthesis of Support Vector Machines: Case Study on ECG Arrhythmia Detection for Xilinx Zynq SoC”. In: *Journal of Signal Processing Systems* 88.2 (Aug. 2017), pp. 127–147. DOI: 10.1007/s11265-017-1230-1.

[75] Junyi Liu, John Wickerson, and George A. Constantinides. “Loop Splitting for Efficient Pipelining in High-Level Synthesis”. In: *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). Washington, DC, USA: IEEE, May 2016, pp. 72–79. DOI: 10.1109/FCCM.2016.27.

[76] Meena Belwal and T.K. Ramesh. “Q-PIR: A quantile based Pareto iterative refinement approach for high-level synthesis”. In: *Engineering Science and Technology, an International Journal* 34 (Oct. 2022), p. 101078. DOI: 10.1016/j.jestch.2021.11.004.

[77] Yiheng Gao and Benjamin Carrion Schafer. “Effective High-Level Synthesis Design Space Exploration through a Novel Cost Function Formulation”. In: *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2021 IEEE International Symposium on Circuits and Systems (ISCAS). Daegu, Korea: IEEE, May 2021, pp. 1–5. DOI: 10.1109/ISCAS51556.2021.9401684.

[78] Gereon Fuhr et al. “Automatic Energy-Minimized HW/SW Partitioning for FPGA-Accelerated MPSoCs”. In: *IEEE Embedded Systems Letters* 11.3 (Sept. 2019), pp. 93–96. DOI: 10.1109/LES.2019.2901224.

[79] Deshya Wijesundera et al. “Wibheda+: Framework for Data Dependency-aware Multi-constrained Hardware-Software Partitioning in FPGA-based SoCs for IoT Applications”. In: *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*. HEART 2018: The 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies. Toronto ON Canada: ACM, June 20, 2018, pp. 1–6. DOI: 10.1145/3241793.3241796.

[80] Wei Zuo et al. “Accurate High-level Modeling and Automated Hardware/Software Co-design for Effective SoC Design Space Exploration”. In: *Proceedings of the 54th Annual Design Automation Conference 2017*. DAC ’17: The 54th Annual Design Automation Conference 2017. Austin TX USA: ACM, June 18, 2017, pp. 1–6. DOI: 10.1145/3061639.3062195.

[81] Adil Iguider et al. “HW/ SW Partitioning Algorithms for Multi- objective Optimization in Embedded Systems”. In: *International Journal of Information Science* (2018), p. 10.

[82] J. Glover, V. Quan, and S. Zolfaghari. “Some new perspectives for solving 0–1 integer programming problems using Balas method”. In: *Computational Management Science* 18.2 (June 2021), pp. 177–193. DOI: 10.1007/s10287-021-00389-6.

[83] Qinglei Zhai et al. “A general approach to solving hardware and software partitioning problem based on evolutionary algorithms”. In: *Advances in Engineering Software* 159 (Sept. 2021), p. 102998. DOI: 10.1016/j.advengsoft.2021.102998.

[84] Tao Zhang et al. “Using Firework Algorithm for Multi-Objective Hardware/Software Partitioning”. In: *IEEE Access* 7 (2019), pp. 3712–3721. DOI: 10.1109/ACCESS.2018.2886430.

[85] Khetatba Mourad and Rachid Boudour. “A Modified Binary Firefly Algorithm to Solve Hardware/Software Partitioning Problem”. In: *Informatica* 45.7 (Nov. 5, 2021). DOI: 10.31449/inf.v45i7.3408.

[86] R.P. Dick, D.L. Rhodes, and W. Wolf. “TGFF: task graphs for free”. In: *Proceedings of the Sixth International Workshop on Hardware/Software Codesign. (CODES/CASHE’98)*. Sixth International Workshop on Hardware/Software Codesign. (CODES/CASHE’98). Seattle, WA, USA: IEEE Comput. Soc, 1998, pp. 97–101. DOI: 10.1109/HSC.1998.666245.

[87] Hao Wang et al. “Hardware-Software Co-Design for Face Recognition on FPGA SoCs”. In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2020 IEEE International Symposium on Circuits and Systems (ISCAS). Seville, Spain: IEEE, Oct. 2020, pp. 1–5. DOI: 10.1109/ISCAS45731.2020.9180922.

[88] Qingcheng Xiao et al. “HASCO: Towards Agile HArdware and Software CO-design for Tensor Computation”. In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). Valencia, Spain: IEEE, June 2021, pp. 1055–1068. DOI: 10.1109/ISCA52012.2021.00086.

[89] Jong Bin Lim and Deming Chen. “Automated Communication and Floorplan-Aware Hardware/Software Co-Design for SoC”. In: *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI). Miami, FL, USA: IEEE, July 2019, pp. 128–133. DOI: 10.1109/ISVLSI.2019.00032.

[90] Georgios Zacharopoulos et al. “RegionSeeker: Automatically Identifying and Selecting Accelerators From Application Source Code”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.4 (Apr. 2019), pp. 741–754. DOI: 10.1109/TCAD.2018.2818689.

[91] Georgios Zacharopoulos et al. “Compiler-Assisted Selection of Hardware Acceleration Candidates from Application Source Code”. In: *2019 IEEE 37th International Conference on Computer Design (ICCD)*. 2019 IEEE 37th International Conference on Computer Design (ICCD). Abu Dhabi, United Arab Emirates: IEEE, Nov. 2019, pp. 129–137. DOI: 10.1109/ICCD46524.2019.00024.

[92] Coen Bron and Joep Kerbosch. “Algorithm 457: finding all cliques of an undirected graph”. In: *Communications of the ACM* 16.9 (Sept. 1973), pp. 575–577. DOI: 10.1145/362342.362367.

[93] Xinheng Liu et al. “High Level Synthesis of Complex Applications: An H.264 Video Decoder”. In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA’16: The 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. Monterey California USA: ACM, Feb. 21, 2016, pp. 224–233. DOI: 10.1145/2847263.2847274.

[94] Maria Kotsifakou et al. “HPVM: heterogeneous parallel virtual machine”. In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’18: 23nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Vienna Austria: ACM, Feb. 10, 2018, pp. 68–80. DOI: 10.1145/3178487.3178493.

[95] Peter Thoman, Herbert Jordan, and Thomas Fahringer. “Adaptive Granularity Control in Task Parallel Programs Using Multiversioning”. In: *Euro-Par 2013 Parallel Processing*. Ed. by Felix Wolf, Bernd Mohr, and Dieter an Mey. Red. by David Hutchison et al. Vol. 8097. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 164–177. DOI: 10.1007/978-3-642-40047-6_19.

[96] Nick Moss, Kei Davis, and Patrick McCormick. “The ARES High-Level Intermediate Representation”. In: *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC). Salt Lake City, UT, USA: IEEE, Nov. 2016, pp. 32–39. DOI: 10.1109/LLVM-HPC.2016.009.

[97] Herbert Jordan et al. “INSPIRE: the insieme parallel intermediate representation”. In: *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT). Edinburgh: IEEE, Oct. 2013, pp. 7–18. DOI: 10.1109/PACT.2013.6618792.

[98] Jieru Zhao et al. “Performance Modeling and Directives Optimization for High-Level Synthesis on FPGA”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.7 (July 2020), pp. 1428–1441. DOI: 10.1109/TCAD.2019.2912916.

[99] Andre B. Perina et al. “Fast Resource and Timing Aware Design Optimisation for High-Level Synthesis”. In: *IEEE Transactions on Computers* 70.12 (Dec. 1, 2021), pp. 2070–2082. DOI: 10.1109/TC.2021.3112260.

[100] Young-Kyu Choi et al. “FLASH: Fast, Parallel, and Accurate Simulator for HLS”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.12 (Dec. 2020), pp. 4828–4841. DOI: 10.1109/TCAD.2020.2970597.

[101] Steve Dai et al. “Fast and Accurate Estimation of Quality of Results in High-Level Synthesis with Machine Learning”. In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). Boulder, CO, USA: IEEE, Apr. 2018, pp. 129–132. DOI: 10.1109/FCCM.2018.00029.

[102] Kenneth O’Neal et al. “HLSPredict: cross platform performance prediction for FPGA high-level synthesis”. In: *Proceedings of the International Conference on Computer-Aided Design*. ICCAD ’18: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN. San Diego California: ACM, Nov. 5, 2018, pp. 1–8. DOI: 10.1145/3240765.3264635.

[103] Young-Kyu Choi and Jason Cong. “HLScope: High-Level Performance Debugging for FPGA Designs”. In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). Napa, CA, USA: IEEE, Apr. 2017, pp. 125–128. DOI: 10.1109/FCCM.2017.44.

[104] Maria Angelica Davila-Guzman et al. “Analytical Model for Memory-Centric High Level Synthesis-Generated Applications”. In: *IEEE Transactions on Computers* 70.12 (Dec. 1, 2021), pp. 2056–2069. DOI: 10.1109/TC.2021.3115056.

[105] Masoud Shahshahani and Dinesh Bhatia. “Resource and Performance Estimation for CNN Models using Machine Learning”. In: *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI). Tampa, FL, USA: IEEE, July 2021, pp. 43–48. DOI: 10.1109/ISVLSI51109.2021.00019.

[106] Lorenzo Ferretti et al. “DB4HLS: A Database of High-Level Synthesis Design Space Explorations”. In: *IEEE Embedded Systems Letters* 13.4 (Dec. 2021), pp. 194–197. DOI: 10.1109/LES.2021.3066882.

[107] Lorenzo Ferretti et al. “Leveraging Prior Knowledge for Effective Design-Space Exploration in High-Level Synthesis”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (Nov. 2020), pp. 3736–3747. DOI: 10.1109/TCAD.2020.3012750.

[108] Lorenzo Ferretti et al. “Graph Neural Networks for High-Level Synthesis Design Space Exploration”. In: *ACM Transactions on Design Automation of Electronic Systems* 28.2 (Mar. 31, 2023), pp. 1–20. DOI: 10.1145/3570925.

[109] Neha Prakriya et al. *LIFT: LLM-Based Pragma Insertion for HLS via GNN Supervised Fine-Tuning*. Apr. 29, 2025. DOI: 10.48550/arXiv.2504.21187. arXiv: 2504.21187[cs].

[110] Zi Wang and Benjamin Carrion Schafer. “Machine Leaming to Set Meta-Heuristic Specific Parameters for High-Level Synthesis Design Space Exploration”. In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020 57th ACM/IEEE Design Automation Conference (DAC). San Francisco, CA, USA: IEEE, July 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218674.

[111] Jorge Castro-Godínez et al. “AxHLS: design space exploration and high-level synthesis of approximate accelerators using approximate functional units and analytical models”. In: *Proceedings of the 39th International Conference on Computer-Aided Design*. ICCAD ’20: IEEE/ACM International Conference on Computer-Aided Design. Virtual Event USA: ACM, Nov. 2, 2020, pp. 1–9. DOI: 10.1145/3400302.3415732.

[112] Sotirios Xydis et al. “SPIRIT: Spectral-Aware Pareto Iterative Refinement Optimization for Supervised High-Level Synthesis”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.1 (Jan. 2015), pp. 155–159. DOI: 10.1109/TCAD.2014.2363392.

[113] Jianyi Cheng, John Wickerson, and George A. Constantinides. “Exploiting the Correlation between Dependence Distance and Latency in Loop Pipelining for HLS”. In: *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. 2021 31st International Conference on Field-Programmable Logic and Applications (FPL). Dresden, Germany: IEEE, Aug. 2021, pp. 341–346. DOI: 10.1109/FPL53798.2021.00066.

[114] Florian Dewald et al. “Improving Loop Parallelization by a Combination of Static and Dynamic Analyses in HLS”. In: *ACM Transactions on Reconfigurable Technology and Systems* 15.3 (Sept. 30, 2022), pp. 1–31. DOI: 10.1145/3501801.

[115] Afonso Canas Ferreira and João M. P. Cardoso. “Graph-Based Code Restructuring Targeting HLS for FPGAs”. In: *Applied Reconfigurable Computing*. Ed. by Christian Hochberger et al. Vol. 11444. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 230–244. DOI: 10.1007/978-3-030-17227-5_17.

[116] Renato Campos and João M.P. Cardoso. “On Data Parallelism Code Restructuring for HLS Targeting FPGAs”. In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). June 2021, pp. 144–151. DOI: 10.1109/IPDPSW52791.2021.00029.

[117] Jason Cong et al. “Bandwidth Optimization Through On-Chip Memory Restructuring for HLS”. In: *Proceedings of the 54th Annual Design Automation Conference 2017*. DAC ’17: The 54th Annual Design Automation Conference 2017. Austin TX USA: ACM, June 18, 2017, pp. 1–6. DOI: 10.1145/3061639.3062208.

[118] Brandon Reagen et al. “MachSuite: Benchmarks for accelerator design and customized architectures”. In: *2014 IEEE International Symposium on Workload Characterization (IISWC)*. 2014 IEEE International Symposium on Workload Characterization (IISWC). Raleigh, NC, USA: IEEE, Oct. 2014, pp. 110–119. DOI: 10.1109/IISWC.2014.6983050.

[119] Ghada Dessouky et al. “Adaptive Dynamic On-chip Memory Management for FPGA-based reconfigurable architectures”. In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014 24th International Conference on Field Programmable Logic and Applications (FPL). Munich, Germany: IEEE, Sept. 2014, pp. 1–8. DOI: 10.1109/FPL.2014.6927471.

[120] Jasmina Vasiljevic and Paul Chow. “Using buffer-to-BRAM mapping approaches to trade-off throughput vs. memory use”. In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014 24th International Conference on Field Programmable Logic and Applications (FPL). Munich, Germany: IEEE, Sept. 2014, pp. 1–8. DOI: 10.1109/FPL.2014.6927469.

[121] Tingyuan Liang et al. “Hi-DMM: High-Performance Dynamic Memory Management in High-Level Synthesis”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (Nov. 2018), pp. 2555–2566. DOI: 10.1109/TCAD.2018.2857040.

[122] Nicholas V. Giamblanco and Jason H. Anderson. “A Dynamic Memory Allocation Library for High-Level Synthesis”. In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 2019 29th International Conference on Field

Programmable Logic and Applications (FPL). Barcelona, Spain: IEEE, Sept. 2019, pp. 314–320. DOI: 10.1109/FPL.2019.00057.

[123] David B. Thomas. "Synthesisable recursion for C++ HLS tools". In: *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP). London, United Kingdom: IEEE, July 2016, pp. 91–98. DOI: 10.1109/ASAP.2016.7760777.

[124] Greg Stitt and Jason Villarreal. "Recursion flattening". In: *Proceedings of the 18th ACM Great Lakes symposium on VLSI*. GLSVLSI08: Great Lakes Symposium on VLSI 2008. Orlando Florida USA: ACM, May 4, 2008, pp. 131–134. DOI: 10.1145/1366110.1366143.

[125] Lars Middendorf, Christophe Bobda, and Christian Haubelt. "Hardware synthesis of recursive functions through partial stream rewriting". In: *Proceedings of the 49th Annual Design Automation Conference*. DAC '12: The 49th Annual Design Automation Conference 2012. San Francisco California: ACM, June 3, 2012, pp. 1207–1215. DOI: 10.1145/2228360.2228583.

[126] Jason Lau et al. "HeteroRefactor: refactoring for heterogeneous computing with FPGA". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20: 42nd International Conference on Software Engineering. Seoul South Korea: ACM, June 27, 2020, pp. 493–505. DOI: 10.1145/3377811.3380340.

[127] Erik H. D'Hollander, Bruno Chevalier, and Koen De Bosschere. "Calling hardware procedures in a reconfigurable accelerator using RPC-FPGA". In: *2017 International Conference on Field Programmable Technology (ICFPT)*. 2017 International Conference on Field Programmable Technology (ICFPT). Melbourne, VIC: IEEE, Dec. 2017, pp. 271–274. DOI: 10.1109/FPT.2017.8280158.

[128] Shane T. Fleming et al. "PushPush: Seamless integration of hardware and software objects via function calls over AXI". In: *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 2015 25th International Conference on Field Programmable Logic and Applications (FPL). London, United Kingdom: IEEE, Sept. 2015, pp. 1–8. DOI: 10.1109/FPL.2015.7294024.

[129] Nupur Sumeet and Manoj Nambiar. "HLS_PRINT: High Performance Logging Framework on FPGA". In: *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. 2021 31st International Conference on Field-Programmable Logic and Applications (FPL). Dresden, Germany: IEEE, Aug. 2021, pp. 390–390. DOI: 10.1109/FPL53798.2021.00081.

[130] Jaume Bosch et al. "Breaking master-slave model between host and FPGAs". In: *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice

of Parallel Programming. San Diego California: ACM, Feb. 19, 2020, pp. 419–420. DOI: 10.1145/3332466.3374545.

[131] Hanchen Ye et al. "ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation". In: *arXiv:2107.11673 [cs]* (Dec. 22, 2021). arXiv: 2107.11673. URL: http://arxiv.org/abs/2107.11673 (visited on 02/17/2022).

[132] Chris Lattner et al. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Seoul, Korea (South): IEEE, Feb. 27, 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[133] Ruizhe Zhao and Jianyi Cheng. "Phism: Polyhedral High-Level Synthesis in MLIR". In: *arXiv:2103.15103 [cs]* (Mar. 28, 2021). arXiv: 2103.15103. URL: http://arxiv.org/abs/2103.15103 (visited on 02/17/2022).

[134] Stéphane Pouget, Louis-Noël Pouchet, and Jason Cong. "Automatic Hardware Pragma Insertion in High-Level Synthesis: A Non-Linear Programming Approach". In: *ACM Transactions on Design Automation of Electronic Systems* 30.2 (Mar. 31, 2025), pp. 1–44. DOI: 10.1145/3711847.

[135] Amir Yazdanbakhsh et al. "AxBench: A Multiplatform Benchmark Suite for Approximate Computing". In: *IEEE Design & Test* 34.2 (Apr. 2017), pp. 60–68. DOI: 10.1109/MDAT.2016.2630270.

[136] Yuko Hara et al. "CHStone: A benchmark program suite for practical C-based high-level synthesis". In: *2008 IEEE International Symposium on Circuits and Systems*. 2008 IEEE International Symposium on Circuits and Systems - ISCAS 2008. Seattle, WA, USA: IEEE, May 2008, pp. 1192–1195. DOI: 10.1109/ISCAS.2008.4541637.

[137] Shelby Thomas et al. "CortexSuite: A synthetic brain benchmark suite". In: *2014 IEEE International Symposium on Workload Characterization (IISWC)*. 2014 IEEE International Symposium on Workload Characterization (IISWC). Oct. 2014, pp. 76–79. DOI: 10.1109/IISWC.2014.6983043.

[138] Lester Kalms, Ariel Podlubne, and Diana Göhringer. "HiFlipVX: An Open Source High-Level Synthesis FPGA Library for Image Processing". In: *Applied Reconfigurable Computing*. Ed. by Christian Hochberger et al. Vol. 11444. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 149–164. DOI: 10.1007/978-3-030-17227-5_12.

[139] *GitHub - karpathy/llama2.c: Inference Llama 2 in one file of pure C*. URL: https://github.com/karpathy/llama2.c (visited on 01/15/2025).

[140] John A Stratton et al. “Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing”. In: (Mar. 19, 2012), p. 12. URL: https://api.semanticscholar.org/CorpusID:497928.

[141] Christian Bienia. “Benchmarking Modern Multiprocessors”. In: (Jan. 2011), p. 153. URL: https://books.google.pt/books?id=gRG-ZwEACAAJ.

[142] Kevin Barker et al. “PERFECT (Power Efficiency Revolution For Embedded Computing Technologies)”. Dec. 2013. URL: https://hpc.pnnl.gov/PERFECT/ (visited on 07/10/2022).

[143] Yuan Zhou et al. “Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs”. In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’18: The 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. Monterey CALIFORNIA USA: ACM, Feb. 15, 2018, pp. 269–278. DOI: 10.1145/3174243.3174255.

[144] João Bispo and João M.P. Cardoso. “Clava: C/C++ source-to-source compilation using LARA”. In: *SoftwareX* 12 (July 2020), p. 100565. DOI: 10.1016/j.softx.2020.100565.

[145] Tiago Santos and Joao M.P. Cardoso. “Automatic Selection and Insertion of HLS Directives Via a Source-to-Source Compiler”. In: *2020 International Conference on Field-Programmable Technology (ICFPT)*. 2020 International Conference on Field-Programmable Technology (ICFPT). Maui, HI, USA: IEEE, Dec. 2020, pp. 227–232. DOI: 10.1109/ICFPT51103.2020.00039.

[146] Pedro Pinto et al. “Aspect composition for multiple target languages using LARA”. In: *Computer Languages, Systems & Structures* 53 (Sept. 2018), pp. 1–26. DOI: 10.1016/j.cl.2017.12.003.

[147] Susan L Graham, Peter B Kessler, and Marshall K McKusick. “gprof: a Call Graph Execution Profiler”. In: *Proceedings of the SIGPLAN ’82 Symposium on Compiler Construction* 17.6 (), pp. 120–126. DOI: 10.1145/800230.806987.

[148] *Xilinx® Runtime (XRT) Architecture — XRT Master documentation*. URL: https://xilinx.github.io/XRT/2025.2/html/index.html (visited on 01/16/2026).

[149] Aaftab Munshi. “The OpenCL specification”. In: *2009 IEEE Hot Chips 21 Symposium (HCS)*. 2009 IEEE Hot Chips 21 Symposium (HCS). Aug. 2009, pp. 1–314. DOI: 10.1109/HOTCHIPS.2009.7478342.

[150] João N. Matos, João Bispo, and Luís Miguel Sousa. “A C Subset for Ergonomic Source-to-Source Analyses and Transformations”. In: *Proceedings of the 16th Workshop on Rapid Simulation and Performance Evaluation for Design*. RAPIDO ’24: Rapid Simulation and Performance Evaluation for Design. Munich Germany: ACM, Jan. 18, 2024, pp. 1–8. DOI: 10.1145/3642921.3642922.

[151] Young-Kyu Choi et al. “In-Depth Analysis on Microarchitectures of Modern Heterogeneous CPU-FPGA Platforms”. In: *ACM Transactions on Reconfigurable Technology and Systems* 12.1 (Mar. 31, 2019), pp. 1–20. DOI: 10.1145/3294054.

[152] *Corinna G. Lee’s Research on Digital Signal Processors*. 1998. URL: https://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html (visited on 04/14/2023).

[153] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. July 19, 2023. DOI: 10.48550/arXiv.2307.09288. arXiv: 2307.09288[cs].

[154] Sravanthi Kota Venkata et al. “SD-VBS: The San Diego Vision Benchmark Suite”. In: *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 2009 IEEE International Symposium on Workload Characterization (IISWC). Oct. 2009, pp. 55–64. DOI: 10.1109/IISWC.2009.5306794.

[155] AMD Xilinx. *Overview • PetaLinux Tools Documentation: Reference Guide (UG1144) • Reader • AMD Technical Information Portal*. PetaLinux Tools Documentation: Reference Guide (UG1144). Nov. 20, 2025. URL: https://docs.amd.com/r/en-US/ug1144-petalinux-tools-reference-guide (visited on 02/26/2026).

[156] Florent Kirchner et al. “Frama-C: A software analysis perspective”. In: *Formal Aspects of Computing* 27.3 (May 2015), pp. 573–609. DOI: 10.1007/s00165-014-0326-7.

[157] McCabe. “A Complexity Measure”. In: *IEEE Transactions on Software Engineering* (1976). DOI: 10.1109/TSE.1976.233837.

[158] DURAN et al. “OmpSs: A PROPOSAL FOR PROGRAMMING HETEROGENEOUS MULTI-CORE ARCHITECTURES”. In: *Parallel Processing Letters* (2011). DOI: 10.1142/S0129626411000151.

[159] Juan Miguel de Haro et al. “OmpSs@FPGA Framework for High Performance FPGA Computing”. In: *IEEE Transactions on Computers* 70.12 (Dec. 2021), pp. 2029–2042. DOI: 10.1109/TC.2021.3086106.

Appendix A

Publications and Demonstrations

The research presented in this thesis has resulted in the following scientific publications:

[1] Tiago Santos, João Bispo, and João M.P. Cardoso. "A CPU-FPGA Holistic Source-To-Source Compilation Approach for Partitioning and Optimizing C/C++ Applications". In: *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT). Vienna, Austria: IEEE, Oct. 21, 2023, pp. 320–322. DOI: 10.1109/PACT58117.2023.00034.

[2] Tiago Santos, João Bispo, and João M. P. Cardoso. "A Flexible-Granularity Task Graph Representation and Its Generation from C Applications (WIP)". In: *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES '24: 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems. Copenhagen Denmark: ACM, June 20, 2024, pp. 178–182. DOI: 10.1145/3652032.3657580.

[3] Tiago Santos et al. "On Improving the HLS Compatibility of Large C/C++ Code Regions". In: *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). ISSN: 2576-2621. May 2025, pp. 282–282. DOI: 10.1109/FCCM62733.2025.00054.

[4] Tiago Santos, João Bispo, and João M. P. Cardoso. "Ph.D. Project: Holistic Partitioning and Optimization of CPU-FPGA Applications Through Source-to-Source Compilation". In: *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). ISSN: 2576-2621. May 2025, pp. 308–309. DOI: 10.1109/FCCM62733.2025.00022.

[5] Tiago Santos et al. "HLS to FPGAs: Extending Software Regions Via Transformations and Offloading Functions to the CPU". In: *2025 IEEE 18th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. 2025 IEEE 18th International

Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC). Singapore, Singapore: IEEE, Dec. 15, 2025, pp. 25–32. DOI: 10.1109/MCSoC67473.2025.00015.

[6] Tiago Santos, João Bispo, and João M. P. Cardoso. "Expanding and partitioning HLS-compatible software regions for CPU-FPGA systems via code transformations". In: *Proceedings of the 16th international symposium on highly efficient accelerators and reconfigurable technologies (HEART 2026)*. Heidelberg, Germany: ACM, June 17, 2026. DOI: 10.1145/3814576.3814597.

[7] Tiago Santos, João Bispo, and João M. P. Cardoso. "Automated holistic HW/SW partitioning and optimization for CPU-FPGA acceleration". In: *Proceedings of the 36th international conference on field-programmable logic and applications (FPL)*. Ghent, Belgium: IEEE, Sept. 9, 2026.

Additionally, the developed compilation flows have been showcased in the following demonstrations:

[1] Tiago Santos, João Bispo, and João M. P. Cardoso. *Holistic Partitioning and Optimization of CPU-FPGA Applications Through Source-to-Source Compilation*. Demonstration presented at the 33rd IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM). Fayetteville, AR, USA, May 2025. URL: https://github.com/specs-feup/clava-fccm-2025-demo (visited on 07/31/2026).

[2] Tiago Santos, João Bispo, and João M. P. Cardoso. *Ergonomic C/C++ source-to-source analysis and transformations for HLS using Clava*. Demonstration presented at the 35th ACM SIGDA / IEEE CEDA University Demonstration at Design Automation Conference (UDDAC). San Francisco, CA, USA, June 2025. URL: https://github.com/specs-feup/clava-dac-2025-demo (visited on 07/31/2026).

[3] Tiago Santos, João Bispo, and João M. P. Cardoso. *Ergonomic C/C++ source-to-source analysis and transformations for HLS using Clava*. Demonstration presented at the GNU Tools Cauldron 2025. Porto, Portugal, Sept. 2025. URL: https://github.com/specs-feup/gnu-tools-cauldron-2025-demo (visited on 07/31/2026).

[4] Tiago Santos, João Bispo, and João M. P. Cardoso. "Ergonomic C/C++ source-to-source analysis and transformations for HLS using Clava". In: *Proceedings of the 36th international conference on field-programmable logic and applications (FPL)*. Ghent, Belgium: IEEE, Sept. 9, 2026.

Appendix B

Extensions to the Clava Source-to-Source Compiler

The extensions developed for the Clava Source-to-Source compiler are open-source repositories found under `https://github.com/specs-feup/`. They are distributed as NPM packages, allowing their reuse in other Clava-based compilation flows, and are as follows:

- clava-code-transforms: this extension contains several code transformations, including array and struct flattening; constant folding and propagation; function outlining, inlining, and voidification; `malloc()` call hoisting; region inlining; and source code amalgamation.
- clava-vitis-integration: allows for the full synthesis, including Place-and-Route, of C/C++ functions from within Clava. It is capable of creating a Vitis HLS project, outputting selected code regions or functions, executing Vitis HLS, and parsing back the synthesis and implementation reports.
- extended-task-graph: our reference ETG implementation is capable of lowering an input application down to the C subset we defined, and generating an ETG from it. Figure C.2 shows the ETG generated for the *disparity* application.
- hoopa: standing for ***ho****listic* ***o****ptimization and* ***p****artitioning* ***a****lgorithms*, this extension implements both our clustering algorithm and our on-chip mapping optimization. Furthermore, it is also capable of creating the XRT boilerplate for CPU-FPGA communication. This extension relies on all other extensions, and can be seen as an orchestrator that can run the entire compilation flow from start to finish.

Appendix C

Additional Definitions and Figures

C.1 An Integer Linear Programming formulation for Hardware/-Software Partitioning

In order to obtain baseline results for a HW/SW partitioning problem, we elect to use a straightforward ILP formulation. We start by reiterating the following properties for each task T_i in a task graph $G(T,C)$, as previously described in Chapter 3:

- Hardware Time HWT_i: the estimated hardware execution time of task T_i, given by an HLS tool;
- Software Time SWT_i: the estimated software execution time of task T_i, measured through profiling the application on a CPU;
- FF Usage FF_i: the number of FPGA Flip-Flops (FFs) required by the hardware implementation of a task T_i;
- LUT Usage LUT_i: the number of FPGA Look-up Tables (LUTs) required by the hardware implementation of a task T_i;
- BRAM Usage $BRAM_i$: the number of FPGA Block Random Access Memories (BRAMs) required by the hardware implementation of a task T_i;
- DSP Usage DSP_i: the number of FPGA Digital Signal Processors (DSPs) required by the hardware implementation of a task T_i;
- Task repetitions R: the number of times the task repeats.

Furthermore, each edge C in a task graph $G(T,C)$ also has a parameter associated with it, the Communication Cost CC_{ij}. This cost represents the cost of communicating data between tasks T_i and T_j, and is calculated by feeding the size of the communicated data to the previously described linear communication cost model. Our task graph formulation specifies that two communication

costs exist: $CPU \rightarrow FPGA$ and $FPGA \rightarrow CPU$. For this basic formulation, though, we assume that these times are the same (i.e., we default to the largest of the two for each edge).

In the ILP formulation itself, we model the HW/SW Partitioning problem as a 0/1 Knapsack Problem, as described in the State-of-the-Art [33, 78–80]: each task T_i has a binary decision variable L_i associated with it, which will be assigned with either a 0 or a 1, depending on whether the task should go to hardware or software. In our formulation, we intend to minimize the overall latency of the application, while also taking into account the cost of communication. Equation C.1 describes this formulation: for each task T_i, the ILP solver will assign a value to L_i so that the sum of task execution times reaches the smallest possible value. Furthermore, for each pair of tasks (T_i, T_j), if L_i is different from L_j (meaning they are assigned to different devices), we add the cost CC_{ij} of communicating data from T_i to T_j.

$$\min \sum_{i=1}^{N} (HWT_i \cdot L_i + SWT_i \cdot (1 - L_i)) \cdot R + \sum_{i=1,j=1}^{N} (CC_{ij} \cdot (L_i \oplus L_j)) \tag{C.1}$$

We also further constrain this ILP formulation by using the constraints $C1$ through $C4$, depicted in equations C.2, C.3, C.4 and C.5. These guarantee that all the tasks offloaded to hardware do not exceed the total number of available resources in the FPGA. They do this by adding up the resource usage of each hardware task, and then checking if it is below the limit. Since on an FPGA we can seldom have optimal resource distribution across multiple kernels (i.e., an ideal scenario where all kernels map into a floorplan with no gaps to spare), we need to reduce the total available resources using a penalty factor of S. This value can be fixed to a value between 0.7 and 0.8, since resource usage above 80% is not realistic in practice.

$$C1 : (\sum_{i=1}^{N} FF_i \cdot L_i) \leq TotalFF \cdot S \tag{C.2}$$

$$C2 : (\sum_{i=1}^{N} LUT_i \cdot L_i) \leq TotalLUT \cdot S \tag{C.3}$$

$$C3 : (\sum_{i=1}^{N} BRAM_i \cdot L_i) \leq TotalBRAM \cdot S \tag{C.4}$$

$$C4 : (\sum_{i=1}^{N} DSP_i \cdot L_i) \leq TotalDSP \cdot S \tag{C.5}$$

$$C5 : T_i \in Def \implies L_i = 0 \tag{C.6}$$

Finally, constraint $C5$, depicted in equation C.6, enforces that all tasks that need to be allocated to software by definition have their decision variable L_i set to 0. This is done by verifying if a given task T_i belongs to the set Def, which contains all the tasks that need to be allocated to software by default. Their partitioning result is known *a priori*, but they must still be part of the problem due to the communication costs between them and the other tasks.

C.2 Additional Graph Figures

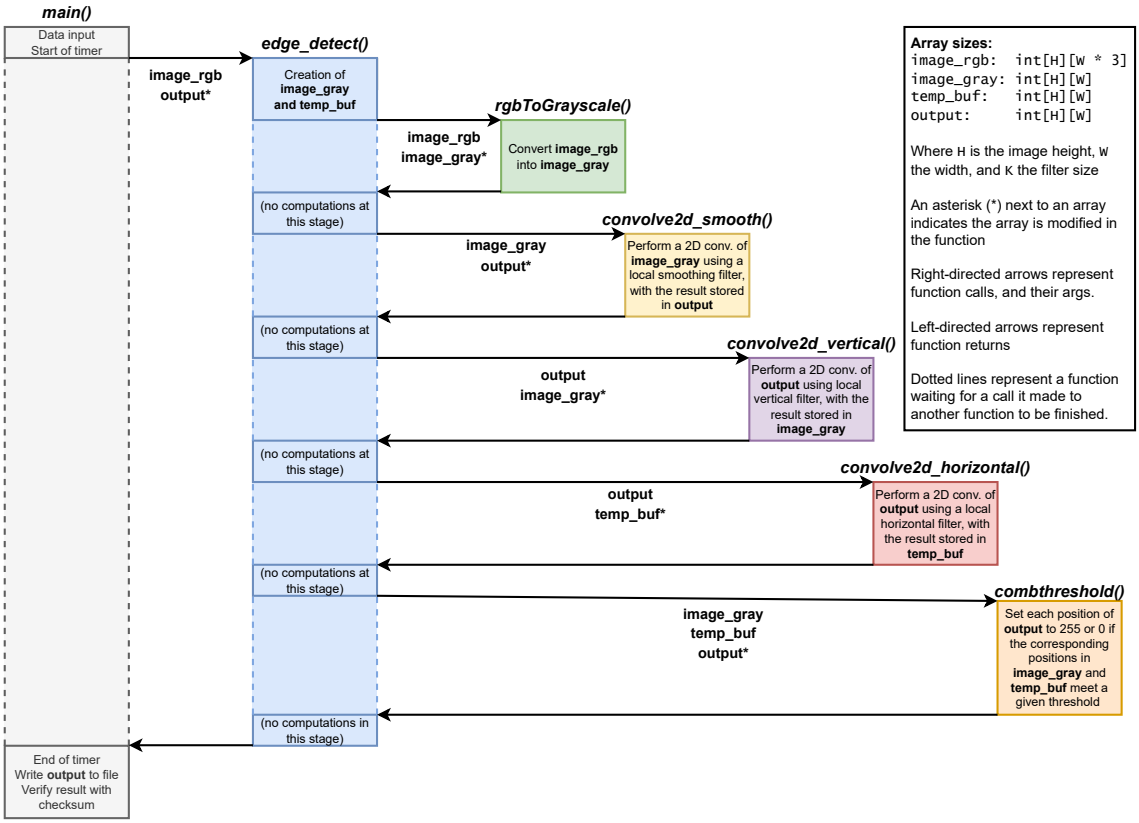


Figure C.1: Function call graph for the ED-O1 version of the *edgedetect* application.

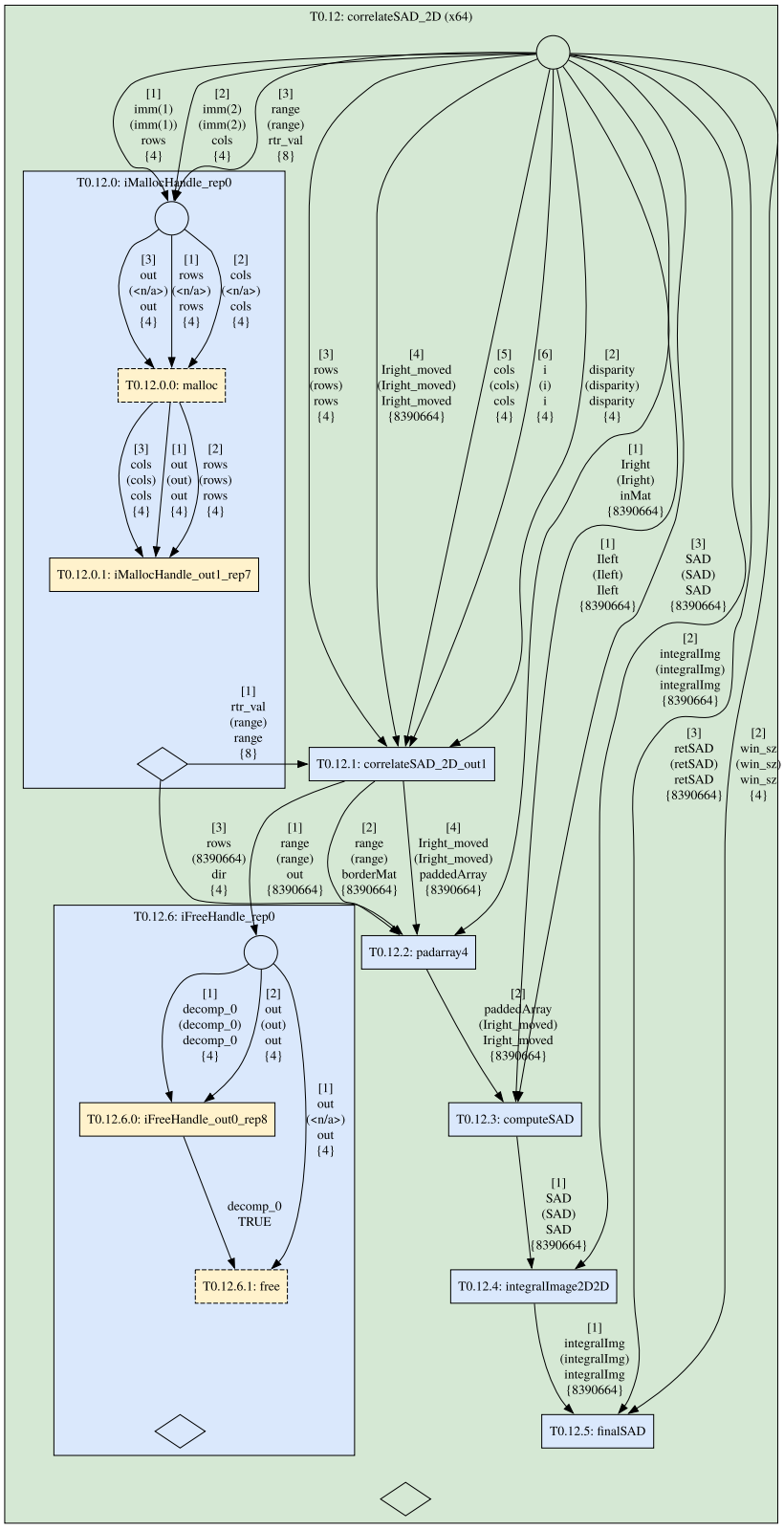


Figure C.2: ETG of the Hotspot Region of *disparity*, with 3 levels of hierarchy. Edges are annotated with the size of the data item communicated, as well as the aliasing information.