COPPE UFRJ
Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia

AUTOMATIC COMPLEX INSTRUCTION IDENTIFICATION WITH HARDWARE SHARING FOR EFFICIENT APPLICATION MAPPING ONTO ASIPS

Alexandre Solon Nery

Tese de Doutorado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Doutor em Engenharia de Sistemas e Computação.

Orientadores: Felipe Maia Galvão França
Nadia Nedjah
Lech Jóźwiak
Henk Corporaal

Rio de Janeiro
Dezembro de 2014

AUTOMATIC COMPLEX INSTRUCTION IDENTIFICATION WITH HARDWARE SHARING FOR EFFICIENT APPLICATION MAPPING ONTO ASIPS

Alexandre Solon Nery

RIO DE JANEIRO, RJ – BRASIL
DEZEMBRO DE 2014

Nery, Alexandre Solon

Automatic Complex Instruction Identification with Hardware Sharing for Efficient Application Mapping onto ASIPs/Alexandre Solon Nery. – Rio de Janeiro: UFRJ/COPPE, 2014.

XVI, 134 p.: il.; 29, 7cm.

Orientadores: Felipe Maia Galvão França
Nadia Nedjah
Lech Jóźwiak
Henk Corporaal

Tese (doutorado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2014.

Bibliografia: p. 110 – 118.

1. Complex Instruction. 2. Maximum Common Subgraph Isomorphism. 3. VLIW-ASIP. I. França, Felipe Maia Galvão *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

AUTOMATIC COMPLEX INSTRUCTION IDENTIFICATION WITH HARDWARE SHARING FOR EFFICIENT APPLICATION MAPPING ONTO ASIPS

Alexandre Solon Nery

Dezembro/2014

Orientadores: Felipe Maia Galvão França
Nadia Nedjah
Lech Jóźwiak
Henk Corporaal

Programa: Engenharia de Sistemas e Computação

Esta tese propõe e discute um novo método eficiente de customização de instruções, juntamente com uma ferramenta que é capaz de automaticamente identificar instruções complexas promissoras para um conjunto relevante de aplicativos de *benchmark*. O método proposto formula o problema de enumeração de subgrafos como um problema de enumeração de *cliques máximos*, com duas novas contribuições: uma no aspecto da conectividade; e a outra no que diz respeito à detecção de (re)-associatividade dos grafos. Os resultados de desempenho da ferramenta proposta para um processador VLIW-ASIP são fornecidos, alcançando um aumento de velocidade de até 54% para a aplicação ray-tracing. Também são apresentados resultados de área do circuito e de consumo de energia das instruções complexas, baseados na tecnologia de 65nm da TSMC. Além disso, esta tese analisa e discute o problema do compartilhamento de hardware no contexto do conjunto de instruções complexas. Embora ferramentas de síntese de hardware disponíveis no mercado sejam capazes de explorar algumas oportunidades de compartilhamento de hardware, esta tese mostra que o resultado é geralmente insatisfatório. Assim, são implementadas e analisadas técnicas de fusão de caminho de dados, atingindo, em média, uma economia de 30% na área de circuito e consumo de energia, para os conjuntos de instruções complexas identificadas nesta tese. Finalmente, arquiteturas multicore são propostas, com base nos processadores extensíveis (ASIPs) usados nesta tese, enriquecidos com o conjunto identificado de instruções complexas e com compartilhamento de hardware. Utilizando até oito ASIPs em paralelo com instruções complexas, uma implementação paralela do algoritmo de ray-tracing é proposta, alcançando até 12× de aceleração em comparação a um único ASIP. As instruções complexas identificadas automaticamente reduzem o tempo de execução em cerca de 36% para a aplicação ray-tracing.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

AUTOMATIC COMPLEX INSTRUCTION IDENTIFICATION WITH HARDWARE SHARING FOR EFFICIENT APPLICATION MAPPING ONTO ASIPS

Alexandre Solon Nery

December/2014

Advisors: Felipe Maia Galvão França
Nadia Nedjah
Lech Jóźwiak
Henk Corporaal

Department: Systems Engineering and Computer Science

Custom instruction identification is an essential part in designing efficient Application-Specific Instruction Set Processors (ASIPs). This thesis proposes and discusses a novel efficient instruction set customization method together with an automatic tool that is able to identify promising custom instruction candidates for a set of relevant benchmark applications. The proposed method formulates the common subgraph enumeration problem as a maximum clique-enumeration problem, with a two-fold novel contribution: one on the connectivity aspect; and the other with respect to the graph (re)-associativity detection. The performance results from the proposed tool for a configurable VLIW-ASIP are provided, achieving a speedup of up to 54% for the ray-tracing application. Circuit area and energy consumption results based on TSMC 65nm technology are also presented. Moreover, this thesis analyzes and discusses the problem of hardware sharing in the context of instruction set customization. Although commercially available hardware synthesis tools are capable of exploiting some hardware sharing opportunities, this thesis shows that the result is usually unsatisfactory. Thus, datapath merging techniques are implemented and analyzed, achieving, on average, substantial circuit area and energy consumption savings of 30% for the sets of custom instructions identified in this thesis. Finally, multi-core architectures are proposed, based on commercially available extensible ASIPs, augmented with the identified set of custom instructions and with hardware sharing optimizations. Using up to eight ASIPs in parallel with complex instructions, a ray-tracer parallel algorithm implementation is proposed, achieving up to 12× speedup in comparison to a single ASIP design. The automatically identified custom instructions provided around 36% execution time reduction for the ray-tracing application.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The recent progress in modern nano-CMOS technology has enabled the implementation of various complex systems on single chips, pushing the development of various kinds of application-specific or domain-specific embedded systems. For many application areas, such embedded systems can be even several orders of magnitude faster, while consuming several times less power than the traditional general purpose CPU-based systems [1]. Increasingly complex and sophisticated embedded systems are required to reliably perform real-time computations to extremely tight schedules with energy and area efficiency never demanded before. The computational demands of high-performance systems for scientific and engineering computation also grow rapidly.

Application-specific processors and hardware accelerators have become an attractive alternative to general-purpose processors [2], as they can be tailored in order to more precisely meet the requirements of modern highly-demanding applications. Specifically, communications and multimedia applications are often very demanding regarding throughput and energy consumption, requiring sophisticated application-specific (co-)processors implementation. For instance, a set of operation patterns that are frequently executed by an application or a class of applications can be implemented as a hardware accelerator or as a set of custom instructions in an ASIP datapath, with a possible substantial increase of the execution speed. Moreover, specific instruction subsets, such as high precision floating-point instructions, may result useless for a given application. Thus, depending on the information collected during an application analysis, these instructions can be either optimized, e.g. changing from high precision floating-point to fixed-point arithmetic [3], or even removed from the processor's datapath, if they are never needed by the given application. This may substantially lower the overall circuit area.

Opportunities created by modern technology can effectively be exploited only through adequate usage of efficient application-specific system architectures and circuit implementations exploiting more adequate concepts of computation, storage and communication. Developing a large digital circuit is a complicated process that involves complex algorithms and a large amount of data. This requires effective and efficient design methods and Electronic Design Automation (EDA) tools for synthesizing the actual hardware platforms implementing the architectures, and for efficient mapping of the applications onto hardware platforms. The ideal scenario would be that human designers only need to develop a high-level behavioral description of the digital system, while the EDA software would be responsible for all the circuit synthesis, placement and routing, being able to produce an optimal circuit implementation automatically. However, from the computational complexity point of view, the synthesis consists of several intractable problems, *i.e.*, problems with no polynomial time solution [4]. Finding the optimal circuit implementation corresponds to a global search for all possible circuit implementation configurations. Therefore, the EDA software must limit the search space of solutions, performing the search on a local basis and applying only a few optimizations and heuristics to guide the direction of the search.

Summing up, the application-specific character of embedded systems and high computation speed, as well as energy and area minimization demands of many modern embedded applications, result in the necessity of effective and efficient application-specific platform for these applications involving application-specific instruction set processors (ASIPs) or hardware accelerators. An adequate design of such application-specific processors and accelerators is thus of particular importance for the Embedded Systems domain. In order to identify and efficiently exploit the acceleration opportunities, as well as guide the design of an area/power efficient architecture and its circuit implementation, a careful and detailed application analysis and exploitation of tradeoffs between the computation speed, circuit area and energy consumption is required.

1.1 Complex instruction identification & Hardware sharing

Usually, most of the development effort in the ASIP design is focused on mapping a given application onto the ASIP itself, while the hardware synthesis tools are left in charge of speed, circuit area and power optimizations. However, the result of such optimizations may often be unsatisfactory, especially because most of the commercially available synthesis tools are only capable of performing a limited set of optimizations, using limited information on a particular design. Therefore, more accurate architecture and circuit optimization methods should be used in parallel to the synthesis tool in order to achieve decent optimization results, especially for complex high-demanding designs.

Ideally, Computer-aided (CAD) tools for application-specific processor design should be able to automatically identify common operation patterns of an application and automatically generate an ASIP with the corresponding embedded custom instructions, either as an (semi-)custom Integrated Circuit (IC) or in a (Re)-Configurable Hardware (e.g. Field-Programmable Gate Array – FPGA). Also, the compiler should be able to automatically schedule such new custom instructions whenever possible. However, currently this is rarely the case, especially because of the problem's inherent complexity. The custom instruction identification process is a subgraph isomorphism problem, which is a well-known computationally complex problem [5–7] and, thus, very time consuming. Most of the existing tools still depend on the designer's expertise to identify custom instructions, implement them in hardware and insert them into the application code. In summary, current CAD tools do not fully automate the instruction set extension process. Designing and manufacturing an ASIP in IC can be a long and expensive process. Still, efficiently automating parts of the design of the application-specific processors can very much help the designer to meet time-to-market requirements.

Instruction Set Customization, also known as Instruction Set Extension (ISE), is a well-known technique to enhance the performance and efficiency of Application-Specific Processors (ASIPs) [8]. For instance, the multiply-accumulate operation pattern is often used in a broad range of multimedia and digital signal processing applications. Therefore, many modern processors already include a multiply-accumulate operation implemented in hardware, resulting in a possible substantial increase of the execution speed and additional circuit area requirements. When several custom instructions are identified for hardware implementation the impact on circuit area is much higher. A naive ad hoc instruction set customization may not result in the required performance improvement, leading to a waste of computing and energy resources. Thus, when performing custom instruction selection, complex tradeoffs between processing speed, circuit area and power consumption must be closely observed.

For those reasons, resource sharing, also known as hardware sharing [4], is a well-known optimization approach traditionally employed for saving circuit area, in which two or more equivalent hardware parts (e.g. functional units) are combined into a single one, provided that they are never going to be used at the same time during an application execution (are mutually exclusive). For instance, in a class of applications there is usually large sets of equivalent patterns (custom instructions) that can be identified, evaluated and finally selected for extending an existing instruction set, which may significantly increase the overall circuit area of a processor. Thus resource sharing represents an important optimization problem for saving circuit area, which can also help to save power [9]. While many synthesis tools provide some resource sharing transformations, their result is often far from ideal [10].

Moreover, this thesis focuses on the extension of a well-known RISC-based ASIP and a commercially available VLIW-based ASIP, both briefly presented in Sections 1.1.1 and 1.1.2. The RISC-based ASIP configuration has been used specifically for the evaluation of the ray-tracing application regarding the custom instruction identification and hardware sharing techniques [11], while the VLIW-based ASIP is the main target processor for the techniques developed in this thesis regarding automatic custom instruction identification [12, 13]. Extensible ASIPs refers to microprocessors that offer the possibility to augment their instruction set architecture with custom instructions, mainly involving its extension through construction of a new application-specific instruction sub-set. In this case, the hardware accelerators or custom functional units implementing the custom instructions are added to the existing processor/core datapath.

1.1.1 Extensible ASIPs: Xilinx MicroBlaze

The Xilinx MicroBlaze is a RISC-based microprocessor described in a Hardware Description Language (HDL) and optimized for synthesis on Xilinx FPGA's (Field-Programmable Gate Arrays). It can be configured in terms of buses, peripherals, number of pipeline stages, support for floating-point operations in hardware, etc. Each microprocessor's instruction set can be extended with up to 16 custom instructions, implemented as co-processors through the Xilinx Fast Simplex Link (FSL) bus, as can be observed in Fig. 1.1.

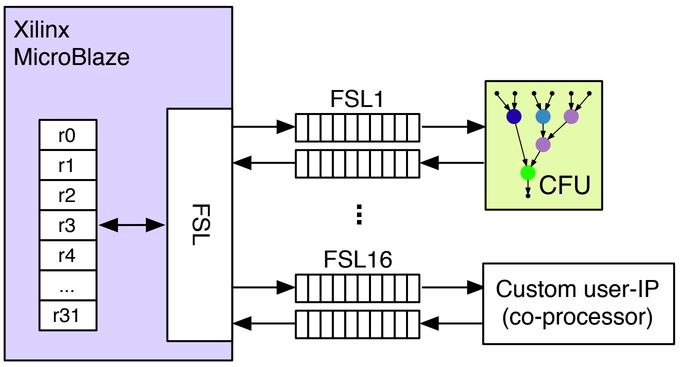


Figure 1.1: Example of a custom instruction and its connection to the microprocessor through the FSL bus. The Custom Function Unit (CFU) attached to the microprocessor acts as a co-processor.

The FSL bus is in fact a pair of FIFO's, where data can be inserted at one end and retrieved at the other end, using the provided *put* and *get* instructions of the microprocessor, respectively. At the custom function unit side, signals must be coded in HDL to sample and store data from/into the FIFO. The FSL latency is of one cycle only. Despite the processor's configurability, the compiler is not retargetable. In that case, each custom instruction need to be mapped manually in the application code in order to be used.

1.1.2 Extensible ASIPs: VLIW-ASIP

The general structure of the extensible VLIW data-path is depicted in Fig. 1.2. It can be configured with any selected kind and number of regular/augmented issue-slots (IS/AIS), function units (FU) and register files (RF). The data-path contains a set of function units organized as a set of parallel issue-slots. In each issue-slot, an operation can be started in every clock cycle. The issue-slots are connected to register files via programmable interconnect and can write to any register file. In general, the function units compute operations on data stored in the register files. The operands of a function unit can also come from a memory port or directly as an immediate value. However, an operation can use at most one immediate.

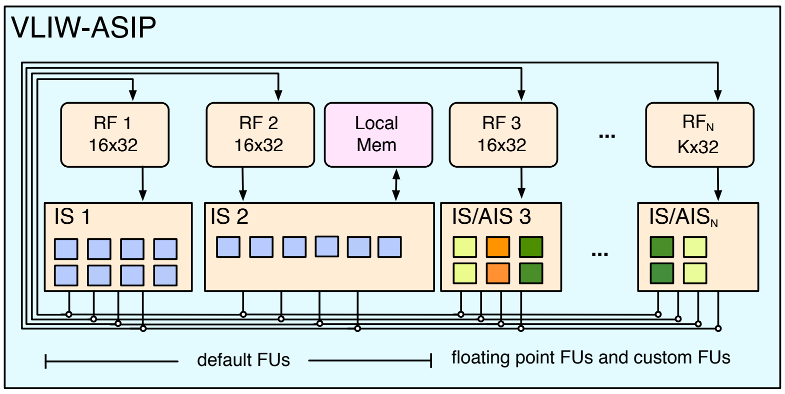


Figure 1.2: VLIW Application-Specific Processor Data-path.

A custom function unit implements a new custom instruction specified by the designer, as shown in Fig. 1.3. An issue-slot with custom function units is an augmented issue-slot (AIS). Due to the configurable nature of this VLIW data-path, its compiler is retargetable. Thus, the compiler is able to automatically recognize the new custom instructions implemented by the custom function units and schedule them whenever possible, based on their time-shape information. Further details are presented in Section 2.3.

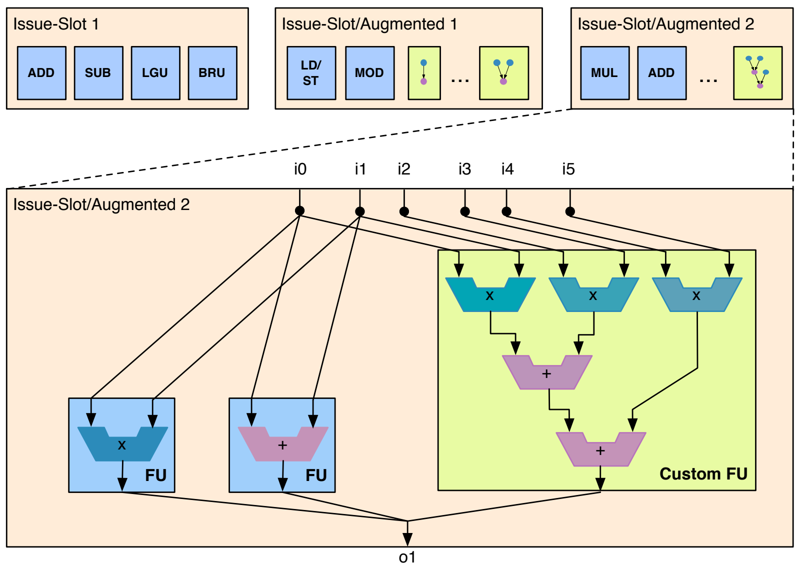


Figure 1.3: VLIW parallel issue-slots and an example of function unit customization, including the *dot product* operation.

1.2 Related work on complex instruction identification

Some of the existing custom instruction identification methods use a template matching approach for extending the instruction set of an ASIP [14], generally because it eliminates the instruction generation step and, thus, simplifies the extension process. Templates are previously defined subgraphs being potential custom instruction candidates, as exemplified in Fig. 1.4. Each template is compared to the application graph to find the set of templates that are matched with the most parts of the application graph.

However, it is easy to observe that this approach is limited to identifying only the custom instructions that are included in the set of pre-defined templates. For instance, in Fig. 1.4, there is a *dot-product* pattern that could be implemented as a custom instruction, possibly yielding a higher speedup. In general, the more closely the custom instructions resemble the most frequent and critical operation patterns of a particular application, the higher speedup and energy gain can be expected. Consequently, the custom instructions should not be pre-decided, but should be decided after a careful analysis of the application [1, 2]. Beyond the template matching approach, complex instruction identification (a.k.a. instruction generation) techniques are often used [15]. The custom instruction identification process is usually formulated as a subgraph isomorphism problem, which is NP-Complete [5]. Therefore, in most of the existing research the instruction set extension problem uses a graph-based methodology to formally describe the problem at hand. For instance, an application is often represented by a Directed Acyclic Graph (DAG), where vertices

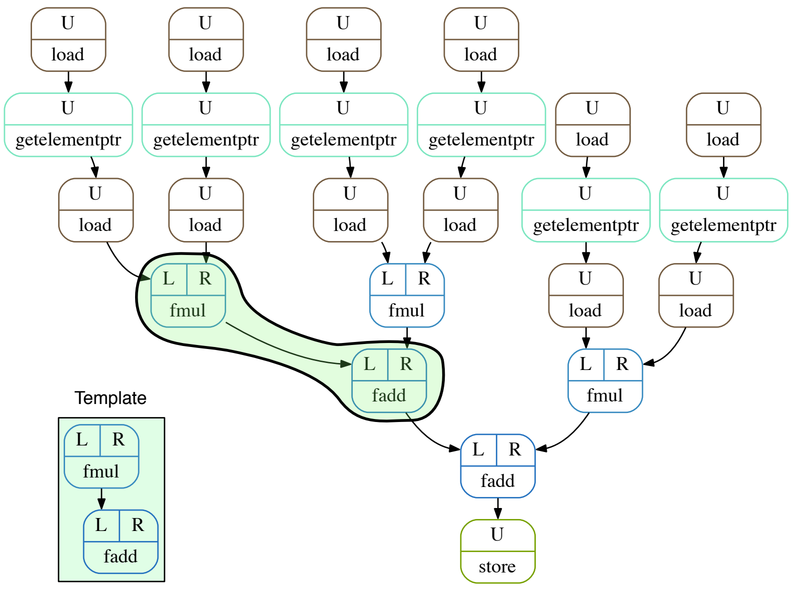
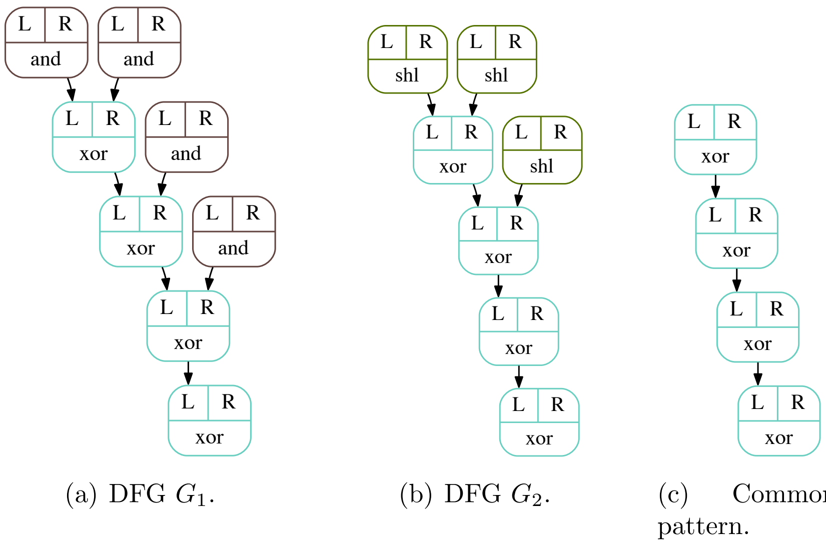


Figure 1.4: Example of a common template pattern (multiply-add) being matched to a Basic Block Data-Flow Graph representation extracted from the Ray-Tracing application.

represent basic operations and the edges represent the data dependencies between the vertices, as shown in Fig. 1.4. Thus, graph theory is commonly used as the standard framework to describe the instruction customization problems.

The custom instruction identification problem boils down to the subgraph isomorphism problem and should not be mistaken with the custom instruction selection problem, which is a graph covering problem [16, 17]. The custom instruction identification process consists of finding the common patterns that may exist between a set of graphs, as depicted in Fig. 1.5.

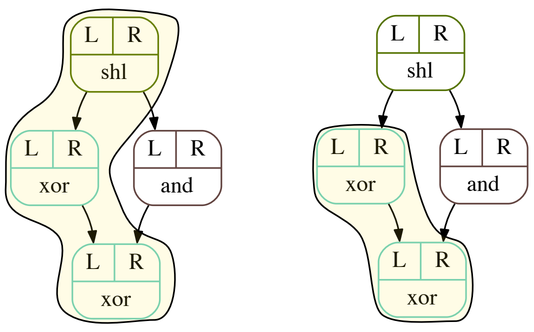


(a) DFG G_1. (b) DFG G_2. (c) Common pattern.

Figure 1.5: Pair of Data-Flow Graphs, G_1 and G_2, together with their corresponding common pattern G_3.

Finding exact solutions can be very time consuming and tedious, because the algorithms have a worse case exponential time complexity. Thus, several approximate algorithms have been proposed [5, 15], including maximum common subsequence algorithms [18–21] for the hardware sharing problem, which is similar to the custom instruction identification problem. However, for a small set of graphs, which is usually the case for custom instruction identification, with basic blocks in the range of hundreds of vertices each, subgraphs may be identified in feasible time, applying pruning strategies to reduce the search space. For instance, micro-architectural constraints regarding the number of input operands and output results can substantially lower the subgraphs identification time [22, 23]. Moreover, memory, branch and floating-point operations are usually forbidden during the pattern identification process, also due to micro-architectural constraints. Thus, only a subset of operations is enabled for custom instruction identification, which helps to further reduce the search space.

Several custom instruction identification methods exists that are based on convex cuts [15, 24–26]. A cut S is an induced subgraph of a graph G, such that S $\subseteq$ G. A cut is convex *iff* there is no path from a vertex $u \in S$ to another vertex $v \in S$ that passes through a vertex $w \notin S$, as shown in Fig. 1.6.



(a) Nonconvex cut. (b) Convex cut.

Figure 1.6: Nonconvex and convex cuts.

Despite of their poor execution time, these methods are often used as the graph-based framework for instruction set extension. They first enumerate all convex cuts (*i.e.*, valid subgraphs) within the data-flow graphs of each basic block of the application. Each subgraph is a potential custom instruction. Enumerating all possible cuts within a basic block is not computationally feasible and, thus, prune strategies must also be used to avoid infeasible regions during the search [27]. For instance, many techniques based on cuts use the number of I/O ports required by the custom instruction function unit and register file to prune the set of subgraphs that can be enumerated, limiting the size and shape of the generated custom instructions. Also, once all the subgraphs have been enumerated, the graph isomorphism algorithm is exhaustively applied to identify which subgraphs are equivalent to one another, finally resulting in the production of a library of equivalent subgraphs, *i.e.*, custom instructions.

The proposed method formulates the common subgraph enumeration problem as a maximum clique-enumeration problem, similar to the method described in [28, 29] for the datapath merging problem. The proposed method differentiates itself from previous approaches in the introduction of graph connectivity and (re)-associativity analysis. The methodology enables a greater control of several aspects of the problem, especially regarding the produced custom instructions granularity, the produced subgraphs connectivity and the (re)-associativity detection. The graph connectivity can impact, for example, on the clique-enumeration execution time, reducing the subgraph isomorphism space-domain to only search for connected graphs [45]. Furthermore, associativity detection further enables the matching of pair of graphs that would be otherwise considered different by traditional graph-matching algorithms that generally deals only with commutativity aspects [19, 28].

In this work, automatic custom instruction identification for a set of well-known benchmark applications is presented, together with performance, circuit area and energy consumption estimation results. The identified custom instructions are implemented in a commercially available extensible VLIW ASIP processor, while most methods published so far have focused on academic/prototype single-issue architectures [15, 23, 25, 30]. Only a few authors have addressed multi-issue architectures [17, 31–34], also based on academic/prototype VLIW architectures. The VLIW architecture has several advantages over traditional single-issue architectures, especially regarding Instruction Level Parallelism (ILP). Thus, this work also presents speedup results for different configurations of augmented issue-slots.

1.3 Related work on hardware sharing

In the part of this thesis related to hardware sharing, most commercially available synthesis tools can perform to a certain degree some area, delay and energy optimizations, based mainly on local information about the design at each synthesis stage. The whole synthesis process consists of several intractable problems, i.e. algorithms with no polynomial-time solution. This is an inherent limitation of every synthesis software and cannot be overcome by faster hardware, especially due to the size and complexity of many modern digital designs [4]. Therefore, the synthesis tools rely on heuristics to produce a sub-optimal solution. While many synthesis tools provide some resource sharing transformations, their result is often far from ideal [10, 35]. The quality and efficiency of a design is still dependent on a human designer's expertise. Still, such area optimization together with automatic custom instruction identification can contribute to speedup the design process of an ASIP and reduce its time-to-market. One important design optimization to save circuit area consists in merging together two or more equivalent parts of a datapath (e.g. function units) that are accessed in a mutually exclusive way. This technique, known as datapath merging, substantially lowers the overall circuit area [18, 19, 21, 36–38].

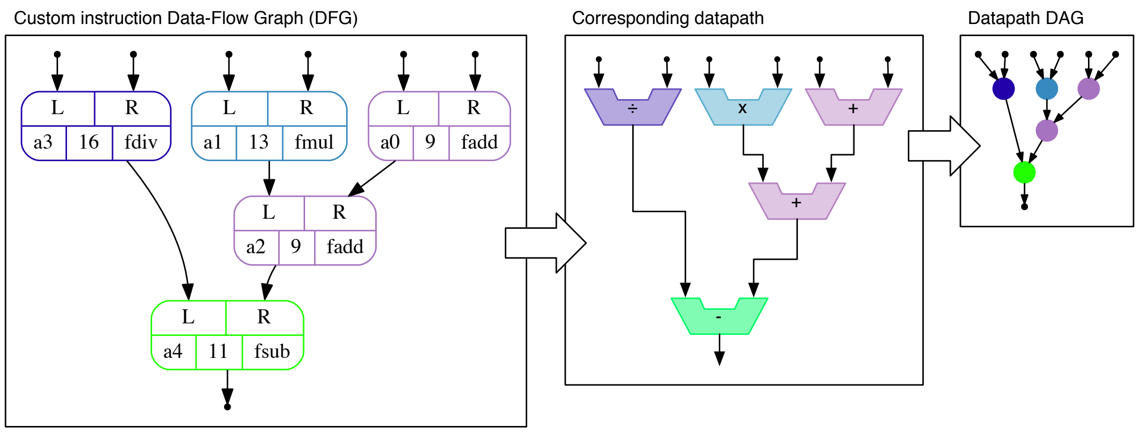
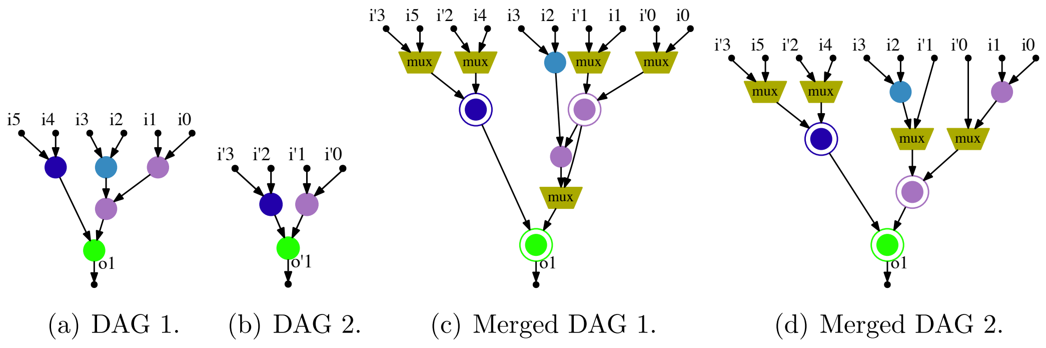


Figure 1.7: Custom instruction Data-Flow Graph (DFG) and its datapath Directed Acyclic Graph (DAG) representation.

Basically, the merging process takes as input the library of identified custom instructions and extracts two of their datapath Directed Acyclic Graphs (DAGs) at a time, *e.g.*, G_1 and G_2, and merges them into a new combined directed acyclic graph G_M that overlaps the equivalent function units and edges of G_1 and G_2. Fig. 1.7 shows an example of a datapath DAG representation. Thereafter, if there are more graphs to merge, the new merged graph G_M is again merged with the next available graph, *e.g.*, G_3, as long as there are equivalent function units and edges to overlap. This process is repeated until all the available graphs have been considered. The final combined datapath must be able to realize the behavior of the original datapaths, as depicted in Fig. 1.8.


i5 i4 i3 i2 i1 i0
o1
i'3 i'2 i'1 i'0
o'1
i'3 i5 i'2 i4 i3 i2 i'1 i1 i'0 i0
mux
o1
i'3 i5 i'2 i4 i3 i2 i'1 i'0 i1 i0
mux
o1

(a) DAG 1. (b) DAG 2. (c) Merged DAG 1. (d) Merged DAG 2.

Figure 1.8: Datapath merging example.

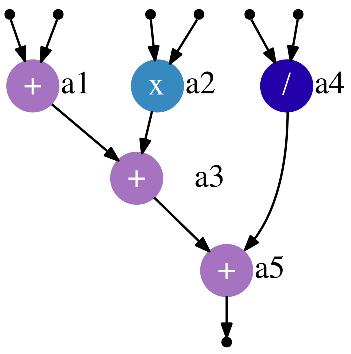
For example, it is possible to observe that the simple example datapath DAGs shown in Fig. 1.8(a) and Fig. 1.8(b) have a few function units in common, because of their equivalent colors. These function units can be merged as highlighted in the merged datapath of Fig. 1.8(c). Notice that before the merging process, both datapaths require 8 function units and 18 interconnections, in total. After merging, the number of the required function units is 5, roughly representing 37.5% reduction, but 5 multiplexers had to be included in order to select which datapath, DAG_1 or DAG_2, should be used. So, even for such simple datapaths, a substantial area reduction is possible.

The main problem with the merged datapaths of Fig. 1.8(c) and Fig. 1.8(d) is that the inserted multiplexers increase the delay of the circuit [19] and, thus, multiplexer inclusion should be avoided whenever possible. The inclusion of multiplexers depends on the number of interconnections needed in the circuit. For instance, the merged DAG illustrated in Fig. 1.8(d) preserves some of the original interconnections and, more importantly, does not include an extra multiplexer. While the total number of interconnections in the merged datapath presented in Fig. 1.8(c) is 21, the total number of interconnections in the merged datapath depicted in Fig. 1.8(d) is 19, which is almost the same as the 18 total interconnections of the original datapaths in Fig. 1.8(a) and Fig. 1.8(b).

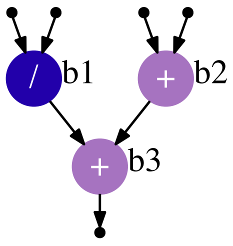
In essence, the datapath merging problem is also a subgraph isomorphism problem, as well as the pattern identification problem. Because of its worse case exponential time complexity, there are several heuristics that try to solve the datapath merging in polynomial time. Let G_1 and G_2 be two directed acyclic graphs of a pair of datapaths, as shown in Fig. 1.9(a) and Fig. 1.9(b), respectively. The first step of the datapath merging process involves the mapping of the vertices or edges of graph G_1 to the equivalent vertices or edges of graph G_2, analogous to the edge mapping process of the custom instruction identification stage. Thus, the equivalent patterns (vertices, edges, etc.) belonging to different DAGs are identified as possible matching. The result is a bipartite graph matching, as shown in Fig. 1.9(c). This matching, in particular, maps with each other all the equivalent vertices that exists between the pair of DAGs that are used as example.

For instance, Fig. 1.9(c) shows all the possible mappings between the vertices of DAGs G_1 and G_2. Observe that one vertex from G_1 cannot be merged to more than one vertex from G_2 at the same time, and vice-versa. Otherwise inconsistencies may be created in the final merged DAG, such as the collapsing of two or more vertices that are data-dependent. Thus, only one of the possible identified mappings will be selected for merging. This mapping selection process will be explained in Section 3.2.2.

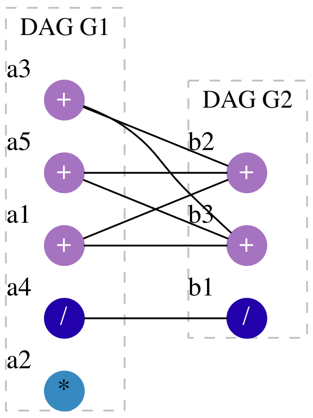
Another possible mapping method, known as *path-based matching*, maps the different graph equivalent patterns based on the *longest common subsequences (or*



(a) Datapath DAG_1.



(b) Datapath DAG_2.



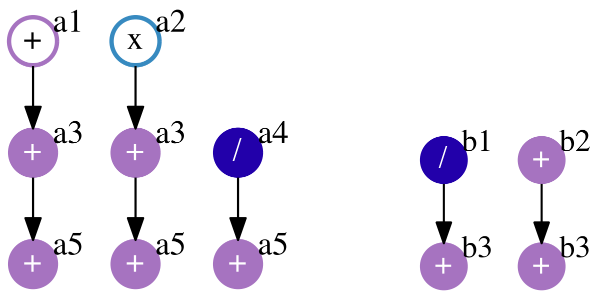
(c) Complete mapping for the pair of DAGs G_1 and G_2.

Figure 1.9: Pair of Datapath DAGs and their mapping of vertices.

substrings) that may exist between paths of the different DAGs, also respecting the order of mapping imposed by the subsequence or substring. The longest common subsequence/substring algorithm is usually applied in the *bioinformatics sequence analysis* context, to discover the longest DNA subsequence/substring that is common to all the DNA sequences that are being compared [39]. The longest common substring must be always contiguous, whereas the longest common subsequence is not necessarily contiguous. In [18], each datapath Directed Acyclic Graph (DAG) is transformed to a set of paths, *i.e.*, subsequences or substrings, that are compared to each other to identify its common parts. Each path is converted into subsequences or substrings, so that the longest common subsequences and longest common substrings algorithms can be used to identify such common parts between the datapaths. Similar resource sharing techniques are used in [19, 21, 38] together with a heuristic algorithm to control the degree of resource sharing during the datapath merging of a set of custom instructions.

In the path-based mapping context, all the possible paths from each datapath DAG are enumerated, in order to produce sequences of vertices that can be represented as strings. The extraction process consists of traversing the DAG from a *source* vertex to a *sink* vertex, and considering all the possible paths in-between these vertices, as in Fig. 1.10, where paths for DAGs G_1 and G_2 from Fig.1.9 are shown. The idea is to break down the datapath DAG into several strings that later can be compared to each other to find, for instance, the longest common subsequences (L-SEQ) or substrings (L-STR).

Thus, a pair-wise comparison is performed between each path from the first DAG and each path from the second DAG, in order to compute the longest common subsequence or the longest common substring existing between the paths from the different DAGs. For instance, in Fig.1.10, the longest common subsequences of vertices are $L - SEQ_1 = (+,+)$ and $L - SEQ_2 = (\sqrt{},+)$. These are coincidentally also the longest common substrings. Based on these informations, matches between vertices of two or more DAGs can be discovered. Indeed, the resulting bipartite graph may contain less mappings, as shown in Fig. 1.11.



(a) Paths from G_1. (b) Paths from G_2.

Figure 1.10: Paths generated from DAGs G_1 and G_2 of Fig. 1.8.

To exemplify the path-based vertex matching using L-SEQ, lets consider the paths in Fig. 1.10. To find the longest common subsequence (L-SEQ) between paths from G_1 (Fig. 1.10(a)) and paths from G_2 (Fig. 1.10(b)), the longest common subsequences between each pair of paths needs to be found, one from G_1 and the other from G_2, *e.g* $\{(+,+),(\sqrt{},+)\}$, and then one of the longest common subsequences is selected, *e.g.* $(\sqrt{},+)$. Consequently, the vertices from G_1 and G_2 corresponding to this L-SEQ $(\sqrt{},+)$ are mapped in the bipartite graph matching, as shown in Fig. 1.11, in the same order that they appear in the longest common subsequence. For instance, A_3 cannot be mapped to B_3, because A_5 appears first in the subsequence analysis, as it can be observed from the paths shown in Fig. 1.10. Notice that following the order given by the L-SEQ, more promising matchings are favored, such as A_5 to B_3, which could have been missed. Thus, additional interconnections in the final merged datapath would have been introduced if A_3 to B_3 had been first naively mapped.

Moreover, computing the longest common subsequence is not always the best choice, because shorter sequences are given lower importance over longer sequences [18]. For instance, in some cases, matching a shorter sequence such as $(\div, *)$ is better than a longer sequence like $(*, +, +)$, because the circuit area of the divider is usually much higher and, hence, merging two dividers together may further reduce the overall circuit area. Therefore, priority can be given to sequences in which the sum of each vertex (function unit) circuit area is the largest, based on the synthesis results shown in Section 2.3.2. Such behavior can be disabled according to the user specification, so that the area information is not used to select the longest common

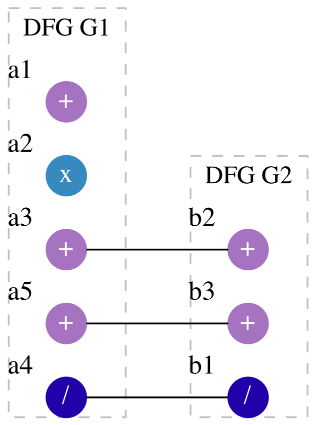


Figure 1.11: Path-based matching for the pair of DAGs G_1 and G_2.

subsequence or substring of maximum area.

Finally, a maximal-clique enumeration heuristic is used in [28, 29, 37] to identify the common parts between a set of datapaths DAGs, including interconnection sharing. Hence, a weighted compatibility graph is built to represent the compatible mappings that exist between the common interconnections of the datapaths. The weights represent the circuit area of each function unit that is going to be merged. Then, the maximal clique with highest weight in the graph represents the largest merging with compatible vertices and edges. This methodology is compared to other methods in Chapter 3 and used for the extension of the custom instruction identification framework, in order to include hardware sharing between the identified custom instructions.

1.4 Benchmark applications

The benchmark applications selected to evaluate the techniques developed in this thesis are presented in this Section. The applications were ported from a mixture of well-known benchmarks, such as MiBench [40] and the ASAM-project benchmarks [41]. Furthermore, additional applications, such as Ray-Tracing, Volume Ray-Casting, Canny Edge Detection and Multi-objective Particle Swarm Optimization were implemented, as they extensively require the use of floating-point computations.

AES The Advanced Encryption Standard (AES) is a specification for the encryption of electronic data, based on the *Rijndael* cipher. It is a symmetric-key algorithm, *i.e.*, it uses the same cryptographic key for encryption of *plaintext* and decryption of *ciphertext*. The tested input data consists of a 305KB text file.

SHA The Secure Hash Algorithm (SHA) is a cryptographic hash function that produces a unique hash value, a.k.a *message digest*, based on an input message. It is often used for generating digital signatures. The input data used for this application is a 3.1MB text file.

CRC32 The 32-bit Cyclic Redundancy Check (CRC32) is an algorithm for error-detection commonly used in digital networks and storage devices. The input data used for this application is a 26MB audio file.

JPEG The JPEG is a very popular lossy compression algorithm for digital images. It uses a lossy form of compression based on the Discrete Cosine Transform (DCT). An image of 600 × 400 pixels was used as input data.

MJPEG The Motion JPEG is a video compression algorithm that compresses each video frame separately as a JPEG image. The input data used for this application are 3 video frames of 128 × 128 pixels.

RT The Ray-Tracing (RT) is a 3D rendering algorithm used for producing high-fidelity images from 3D scenes. In this work, the number of primary-rays fired towards the 3D scene is of 240000 rays, with up to 1 level of reflection/refraction. The 3D scene consists of 2 spheres, 2 rectangles and a cylinder. The size of the produced image is of 600 × 400 pixels.

VRC The Volume Ray-Casting (VRC) is a 3D rendering algorithm often used to produce a medical image from a volumetric dataset acquired by Computer Tomography (CT). In this work, the number of primary-rays fired towards the scene is of 240000 rays and the volumetric dataset has 256^3 voxels. The size of the produced image is of 600 × 400 pixels.

EDGE The canny edge detector (EDGE) is an algorithm for detecting edges in images. The edges are defined as points of an image where the image brightness changes sharply. Edge detection is often used in the fields of machine vision and computer vision. The input data used for this application is an image of 256 × 256 pixels.

MPSO The Multi-objective Particle Swarm Optimization (PSO) is a metaheuristic that tries to find an (sub)-optimal solution for an optimization problem, with regard to more than one objective function. At each iteration, populations of candidate solutions, a.k.a particles, are moved around the search-space. This is expected to move the particles, a.k.a the swarm, to the best solutions. The simulation used in this work creates a swarm of 200 particles and iterates 10 times over 16 dimensions.

1.5 Contributions

This thesis presents novel techniques in the field of automatic custom instruction identification, which enables the design of more efficient Application-Specific Instruction Set Processors. The main contributions are listed bellow:

- A complete automatic custom instruction identification framework and tool that starts with an application described in C and produces a library of potential custom instruction candidates, which can be implemented into the ASIP datapath in order to speedup the execution of the corresponding application.
- A novel graph-matching algorithm is proposed and implemented to identify common patterns, *i.e.*, custom instructions. The tool formulates the subgraph isomorphism problem as a maximum clique-enumeration problem. A novel Connectivity Graph is proposed in order to prune the Compatibility Graph search space, producing only connected subgraphs in less time. Also, original (re)-associativity detection techniques are implemented, potentially enabling the identification of subgraphs that would be otherwise considered different by traditional exact graph-matching algorithms.
- An original analysis of coarse-grain and fine-grain custom instructions is presented. Each (maximum)-clique corresponds to a common subgraph and the tool can automatically identify common operation patterns of different granularities between a set of basic blocks extracted from an application, or class of applications. The size of each enumerated clique determines the granularity of the corresponding generated custom instruction.
- The important problem of hardware reuse in synthesis of ASIPs and hardware accelerators implemented in modern nano-CMOS technologies is presented. This work analyses the problem and presents important results of high practical relevance from extensive experimental research regarding this issue. It demonstrates that the state-of-the-art automatic synthesis tools do not address this problem in a sufficient way, and shows how important an effective hardware reuse is for an adequate synthesis of ASIPs and hardware accelerators.
- A hardware reuse optimization phase that analyzes the function units of the custom instructions and merges their common patterns (functional units). The

proposed optimization demonstrates the big influence of hardware reuse on all the major physical parameters: computation speed, circuit area and energy consumption. In the part of the research related to this issue that is reported in this thesis, it focused on resource sharing of custom instructions that reside in the same issue-slot of the VLIW-ASIP. Therefore, these custom instructions are mutually exclusive (are not going to be executed at the same time) and can be shared.

- An efficient MPSoC with custom instruction speedups that achieves up to 12× speedup when eight extensible ASIPs are used in parallel in respect to a single ASIP design. The automatically identified custom instructions provided around 36% execution time reduction for the ray-tracing application. Circuit area and energy consumption estimations are also provided for the set of identified custom instructions.

1.6 Thesis outline

This thesis is organized as follows: Chapter 2 proposes and discusses the Instruction Customization Framework, with emphasis on maximal-clique enumeration subgraph isomorphism. Then, Chapter 3 describes many aspects related to hardware sharing at the RTL synthesis and proposes an extension to the Instruction Customization Framework based on datapath merging techniques. Chapter 4 proposes custom multimedia parallel architectures based on multiple ASIPs and an ASIP-based MPSoC with complex instruction speedup, which are usually employed to speedup computing-intensive kernels of highly-demanding applications or class of applications. Finally, Chapter 5 draws the conclusion of this thesis and discusses some ideas for future work.

Chapter 2

Instruction Customization Framework

In this chapter, the automated custom instruction identification framework is presented. The design flow of the framework is depicted in Fig. 2.1. It is subdivided into the following three main stages: (i) Profiling, (ii) Custom Instruction Identification and (iii) ASIP Implementation. Details about each stage are presented in the following Sections.

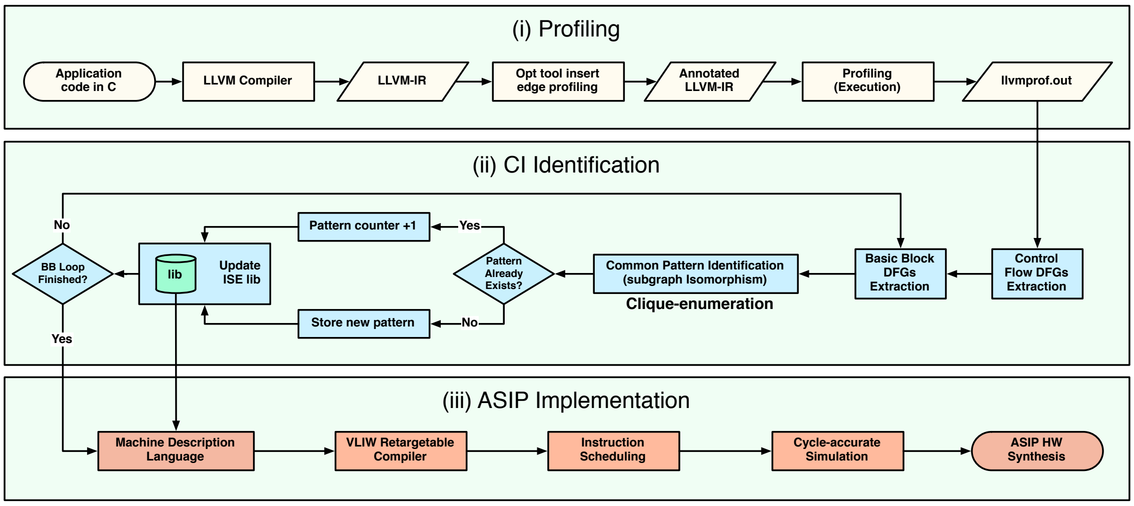
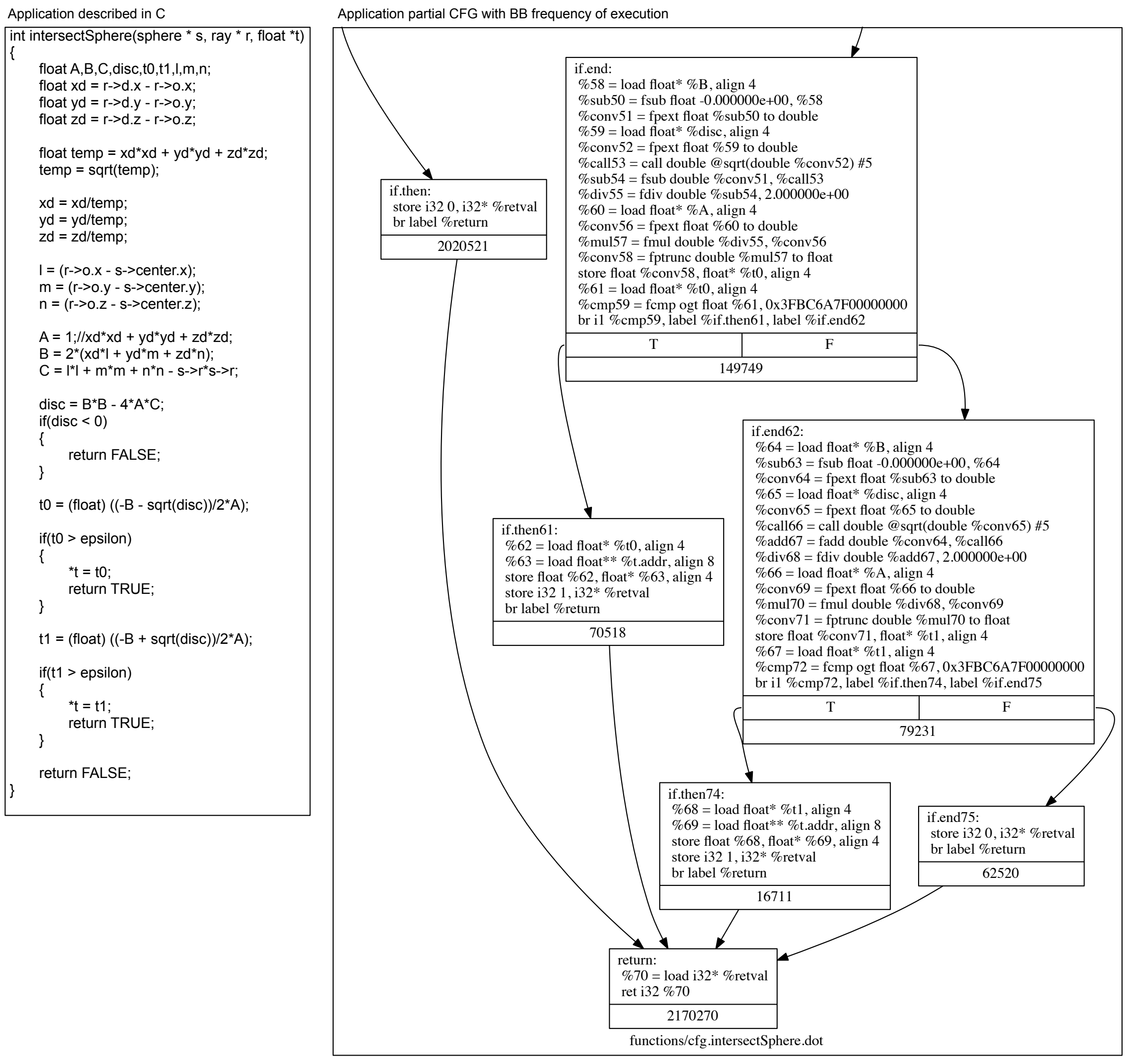


Figure 2.1: Instruction Set Customization Design Flow.

2.1 Application profiling

The Profiling stage is performed using the LLVM compiler infrastructure [42]. It starts with an application described in C. Then, the application code is compiled using LLVM Clang compiler, which produces an Intermediate Representation (LLVM-

IR) of the application code. Before the application is executed, instrumentation code for edge profiling is inserted into the LLVM-IR code by the LLVM *opt* tool, so that information on the execution frequency of each basic block is collected during the application run-time. Thus, the annotated LLVM-IR is executed, producing the basic block profiling information. For the sake of simplicity and readability, Fig. 2.2 shows only a part of the resultant Control Flow Graph (CFG) of an application, with each Basic Block[1] (BB) containing the corresponding code in LLVM-IR and annotated with its frequency of execution. At the end, the result of the Profiling stage is transferred to the Custom Instruction (CI) identification stage.


Application described in C
Application partial CFG with BB frequency of execution
functions/cfg.intersectSphere.dot

```
int intersectSphere(sphere * s, ray * r, float *t)
{
      float A,B,C,disc,t0,t1,l,m,n;
      float xd = r->d.x - r->o.x;
      float yd = r->d.y - r->o.y;
      float zd = r->d.z - r->o.z;

      float temp = xd*xd + yd*yd + zd*zd;
      temp = sqrt(temp);

      xd = xd/temp;
      yd = yd/temp;
      zd = zd/temp;

      l = (r->o.x - s->center.x);
      m = (r->o.y - s->center.y);
      n = (r->o.z - s->center.z);

      A = 1;//xd*xd + yd*yd + zd*zd;
      B = 2*(xd*l + yd*m + zd*n);
      C = l*l + m*m + n*n - s->r*s->r;

      disc = B*B - 4*A*C;
      if(disc < 0)
      {
            return FALSE;
      }

      t0 = (float) ((-B - sqrt(disc))/2*A);

      if(t0 > epsilon)
      {
            *t = t0;
            return TRUE;
      }

      t1 = (float) ((-B + sqrt(disc))/2*A);

      if(t1 > epsilon)
      {
            *t = t1;
            return TRUE;
      }

      return FALSE;
}
```

```
if.then:
 store i32 0, i32* %retval
 br label %return
2020521
```

```
if.end:
 %58 = load float* %B, align 4
 %sub50 = fsub float -0.000000e+00, %58
 %conv51 = fpext float %sub50 to double
 %59 = load float* %disc, align 4
 %conv52 = fpext float %59 to double
 %call53 = call double @sqrt(double %conv52) #5
 %sub54 = fsub double %conv51, %call53
 %div55 = fdiv double %sub54, 2.000000e+00
 %60 = load float* %A, align 4
 %conv56 = fpext float %60 to double
 %mul57 = fmul double %div55, %conv56
 %conv58 = fptrunc double %mul57 to float
 store float %conv58, float* %t0, align 4
 %61 = load float* %t0, align 4
 %cmp59 = fcmp ogt float %61, 0x3FBC6A7F00000000
 br i1 %cmp59, label %if.then61, label %if.end62
T | F
149749
```

```
if.then61:
 %62 = load float* %t0, align 4
 %63 = load float** %t.addr, align 8
 store float %62, float* %63, align 4
 store i32 1, i32* %retval
 br label %return
70518
```

```
if.end62:
 %64 = load float* %B, align 4
 %sub63 = fsub float -0.000000e+00, %64
 %conv64 = fpext float %sub63 to double
 %65 = load float* %disc, align 4
 %conv65 = fpext float %65 to double
 %call66 = call double @sqrt(double %conv65) #5
 %add67 = fadd double %conv64, %call66
 %div68 = fdiv double %add67, 2.000000e+00
 %66 = load float* %A, align 4
 %conv69 = fpext float %66 to double
 %mul70 = fmul double %div68, %conv69
 %conv71 = fptrunc double %mul70 to float
 store float %conv71, float* %t1, align 4
 %67 = load float* %t1, align 4
 %cmp72 = fcmp ogt float %67, 0x3FBC6A7F00000000
 br i1 %cmp72, label %if.then74, label %if.end75
T | F
79231
```

```
if.then74:
 %68 = load float* %t1, align 4
 %69 = load float** %t.addr, align 8
 store float %68, float* %69, align 4
 store i32 1, i32* %retval
 br label %return
16711
```

```
if.end75:
 store i32 0, i32* %retval
 br label %return
62520
```

```
return:
 %70 = load i32* %retval
 ret i32 %70
2170270
```

Figure 2.2: A ray-tracing intersection function code in C and its partially represented Control-Flow Graph (CFG). Each basic block code is represented using the LLVM-IR.

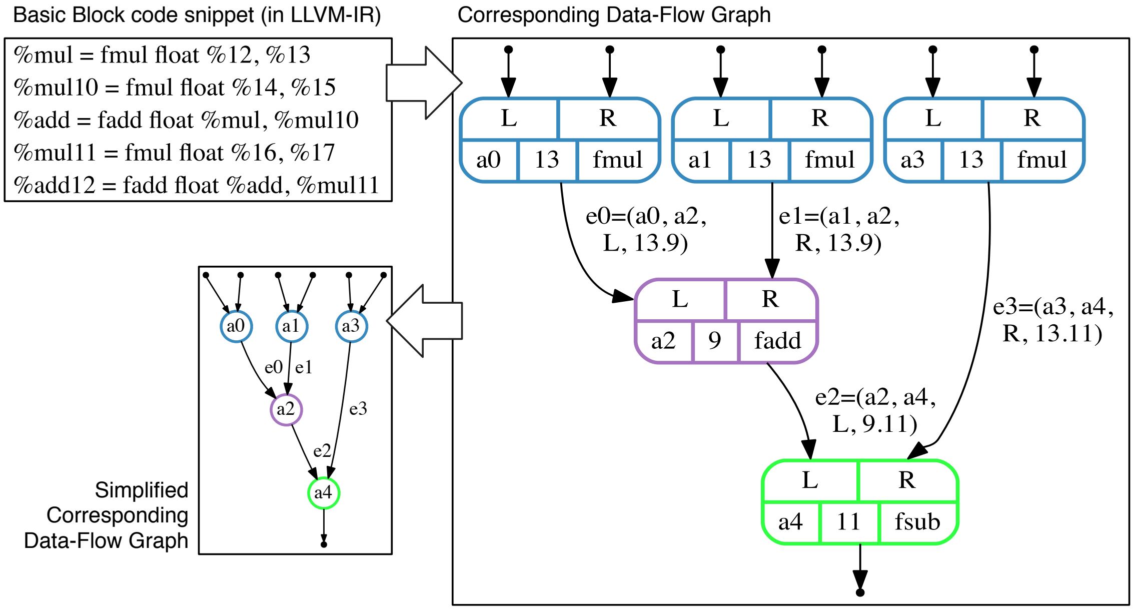
[1]A Basic Block is a portion of an application code with only one entry and only one exit point.

2.2 Custom instruction identification

The Custom Instruction (CI) identification stage of the framework design flow corresponds to one of the main contributions of this thesis. It starts with the extraction of Control Flow Graphs (CFGs) from the profiled application stage and also the extraction of the Data-Flow Graphs (DFGs) of each Basic Block (BB).

Definition 2.1. Each basic block is a directed acyclic graph, also known as data-flow graph, $G = (V, E)$, where each vertex $v \in V = \{v_1, v_2, \cdots, v_n\}$ corresponds to a basic operation of the basic block, such that $v = (id, c, op)$ indicated the id of the basic operation op with color $c \in \mathbb{N}$. Each directed edge $e \in E = \{e_1, e_2, \cdots, e_n\}$ corresponds to a dependency between a pair of basic operations, such that $e = (u, v, p, f)$ indicates a data transfer from vertex u to vertex v, through port $p \in \{L, R, U\}$ and with color $f \in \mathbb{R}$. Each port letter stands for (L) Left, (R) Right and (U) Unary. The color and port information of the edges are commonly not shown, such that $e = (u, v)$ is often used to simply indicate a data transfer from vertex u to vertex v.

Example 2.2. In Fig. 2.3, vertex a_0 corresponds to a floating-point multiplication operation (*fmul*), while vertex a_2 corresponds to a floating-point addition operation (*fadd*). Edge $e_0 = (a_0, a_2, L, 13.9)$ represents the transfer of the *fmul* operation result to the left input port of the *fadd* operation.



```
%mul = fmul float %12, %13
%mul10 = fmul float %14, %15
%add = fadd float %mul, %mul10
%mul11 = fmul float %16, %17
%add12 = fadd float %add, %mul11
```

Figure 2.3: A basic block code snippet example together with its Data-Flow Graph representation. A simplified version of the data-flow graph is also depicted.

Definition 2.3. Each vertex $v \in V$ is given a unique color number $c \in \mathbb{N}$, corresponding to the basic operation it represents, *i.e.*, based on the code of the LLVM-IR it represents. The color f of each edge $e = (u, v, p, f)$ is the concatenation of the source vertex u color number, the decimal point and the target vertex v color number, yielding a color $f \in \mathbb{R}$. Thus, the source vertex color number becomes the integer-part of the edge color, while the target vertex color number becomes the fractional-part of the edge color. The color information is used to identify the equivalent vertices or edges that may exist between a pair of graphs.

Example 2.4. Back to Fig. 2.3, vertex a_0 corresponds to the *fmul* operation, with color number $c = \{13\}$, and vertex a_2 corresponds to the *fadd* operation, with color number $c = \{9\}$. These colors correspond to the LLVM-IR code of their respective operations *fmul* and *fadd*. The edge color from a_0 to a_2 is $f = \{13.9\}$.

In the following Sections, the Custom Instruction (CI) pattern identification methodology is presented. Given a pair of data-flow graphs, i.e., directed acyclic graphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, such that $|V_1| \geq |V_2|$, the objective is to enumerate all the patterns of G_1 (*i.e.*, subgraphs) that are isomorphic to patterns of G_2. While the problem of deciding whether a subgraph of G_1 is isomorphic to a subgraph of G_2 is NP-Complete, the problem of enumerating all the common patterns is a well-known NP-Hard problem, known as the Maximum Common Subgraph Isomorphism problem (MCS). The CI pattern identification methodology uses the *igraph* library [43] for the specification of the MCS problem.

2.2.1 CI pattern identification

The whole common pattern enumeration process can be divided in several smaller steps: (i) mapping of common edges, (ii) construction of a compatibility graph, (iii) construction of a connectivity graph, (iv) intersection and enumeration of maximum cliques.

2.2.1.1 Mapping of common edges

The first step of the CI Pattern Identification process consists of identifying all the common edges that may exist between a pair of graphs G_1 and G_2. Thus, all the common edges need to be mapped, *i.e.*, associated to each other. In order to fully understand the mapping, let us consider the following definition:

Definition 2.5. Let e_i and e_j be any edge of G_1 and G_2, respectively. An edge mapping is the association of two edges $e_i \in E_1$ and $e_j \in E_2$, such that e_i and e_j have the same color f and the same port p information.

Example 2.6. In Fig. 2.4, all the edge mappings are represented by the dashed lines and labeled according to the given edge *id*. For instance, the mapping e_8/e'_5 associates edge $e_8 = (a_8, a_9, U, 48.37)$ of G_1 with edge $e'_5 = (b_5, b_6, U, 48.37)$ of G_2, because they have the same color $f = 48.37$ and the same port $p = U$.

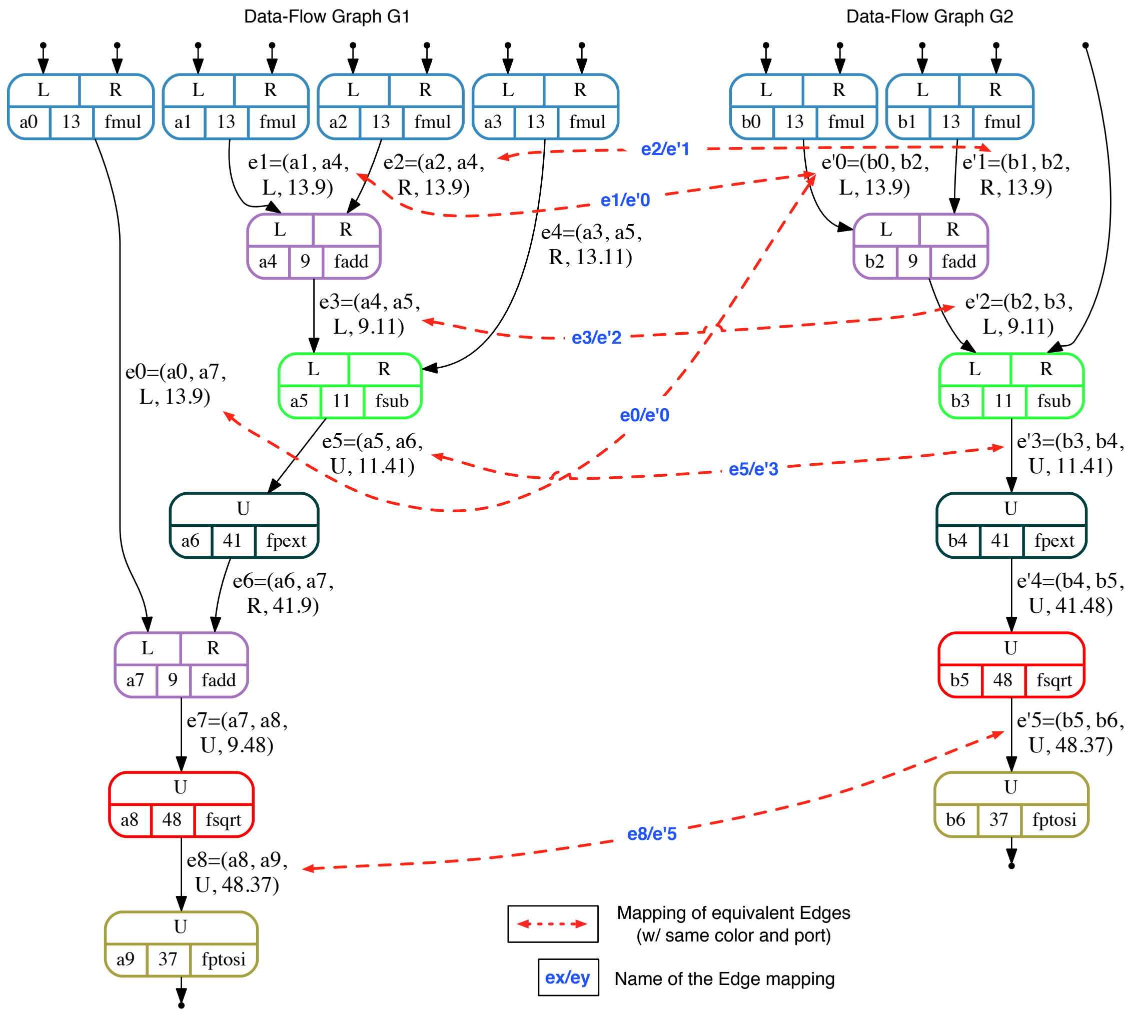


Figure 2.4: A pair of Data-Flow Graphs and their mapping of equivalent edges, which are represented by red dashed arrows and labeled according to the given edges id. For instance, edge $e_8 = (a_8, a_9, U, 48.37)$ of G_1 is mapped to edge $e'_5 = (b_5, b_6, U, 48.37)$ of G_2, because they have the same color $f = 48.37$ and enter through the same port $p = U$.

2.2.1.2 Compatibility graph

Once all the equivalent edges of G_1 are associated to the equivalent edges of G_2, *i.e.*, the ones with the same color and f and same port p, the compatibility graph can be built in order to identify the set of edge mappings that are compatible with each other.

Definition 2.7. Let $e_i = (u_i, v_i)$ and $e_x = (u_x, v_x)$ be two edges belonging to data-flow graph G_1, while let $e'_j = (u_j, v_j)$ and $e'_y = (u_y, v_y)$ be two edges belonging to data-flow graph G_2. Two distinct edge mappings, such as e_i/e'_j and e_x/e'_y, are compatible to one another *iff* $e_i \neq e_x$ and $e'_j \neq e'_y$. Moreover, the source and target vertices corresponding to edges represented by these two edge mappings must be different, such that there is no vertex of G_1 being mapped to two vertices of G_2, or vice-versa.

Example 2.8. Consider the pair of data-flow graphs G_1 and G_2 depicted in Fig. 2.4, and their edge mappings. It can be noted that the edge mappings e_1/e'_0 and e_0/e'_0 are not compatible to each other, because edges e_1 and e_0 belonging to data-flow graph G_1 are being mapped to the same edge e'_0 of data-flow graph G_2. A more subtle example can be observed from the edge mappings e_3/e'_2 and e_0/e'_0, where even though $e_3 \neq e_0$ and $e'_2 \neq e'_0$, their corresponding vertices are not different, in such a way that vertices a_4 and a_7 are mapped to the same vertex b_2.

The compatibility criterion is used to forbid inconsistencies when producing a subgraph corresponding to a group of edge mappings. Otherwise, such inconsistencies could result in a subgraph which is isomorphic to a subgraph of G_1, but not isomorphic to a subgraph of G_2, or vice-versa. The compatibility graph is defined as follows:

Definition 2.9. A compatibility graph $G_c = (V_c, E_c)$, corresponding to the pair of data-flow graphs G_1 and G_2, is a simple undirected graph, where each vertex $v_i \in V_c$ corresponds to an edge mapping e_i/e'_j. There is an edge $e_i = (u_i, v_i) \in E_c$ *iff* the mappings represented by u_i and v_i are compatible.

Example 2.10. Consider the pair of data-flow graphs G_1 and G_2 depicted in Fig. 2.4, together with their corresponding compatibility graph depicted in Fig 2.5(a). It can be noted that all the edge mappings presented in Fig. 2.4 have become vertices of the compatibility graph. The edges of the compatibility graph represent which edge mappings are compatible to each other. If two vertices of the compatibility graph are not connected by an edge, it means that they are incompatible with each other and, thus, cannot be grouped together to form a subgraph.

Enumerating cliques from the compatibility graph enables the extraction of subgraphs that may exist in the pair of data-flow graphs G_1 and G_2 that were used to construct the compatibility graph in the first place. In graph theory, a clique is a subset of vertices $C \subseteq V$, such that every two vertices in C are connected [44]. In other words, the subgraph induced by C is complete. The maximal clique is a clique that is not included in a larger clique, i.e., it cannot be enlarged by including one more adjacent vertex. The maximum clique is a clique of maximum possible

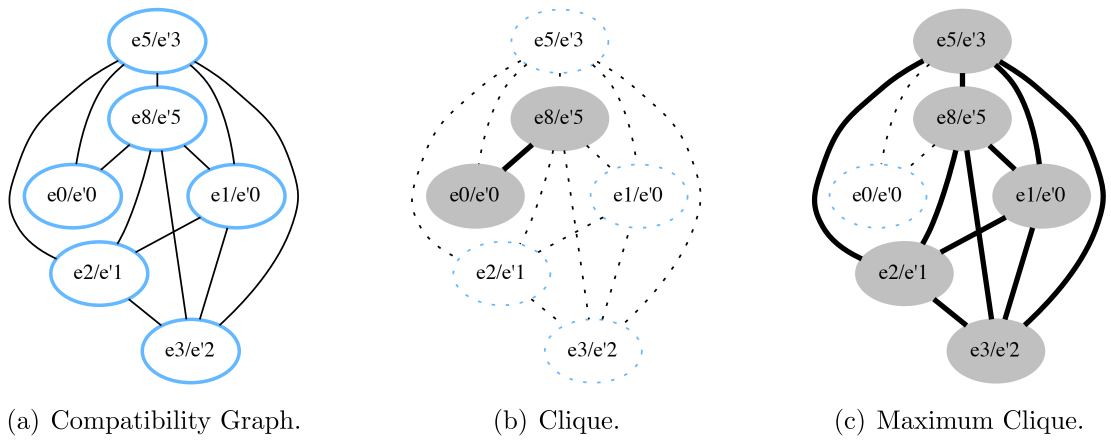


Figure 2.5: Compatibility Graph generated from the mapping of equivalent edges shown in Fig. 2.4, together with a regular enumerated clique and a maximum enumerated clique, which is also a maximal clique.

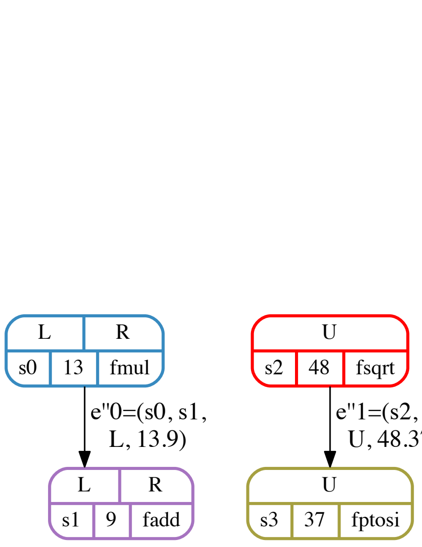
size. Every maximum clique is also a maximal clique, but not all maximal clique is a maximum clique. Thus, the clique enumerated in Fig. 2.5(b) corresponds to the subgraph presented in Fig. 2.6(a), while the maximum clique enumerated in Fig. 2.5(c) corresponds to the subgraph presented in Fig. 2.6(b). Observe that both subgraphs are not weakly connected directed graphs, *i.e.*, it is not possible to reach any vertex starting from any other vertex, regardless of the direction they point. For instance, in the subgraph depicted in Fig. 2.6(a), it is not possible to reach vertex s_3 starting from vertices s_0 or s_1, or vice-versa.

2.2.1.3 Connectivity graph

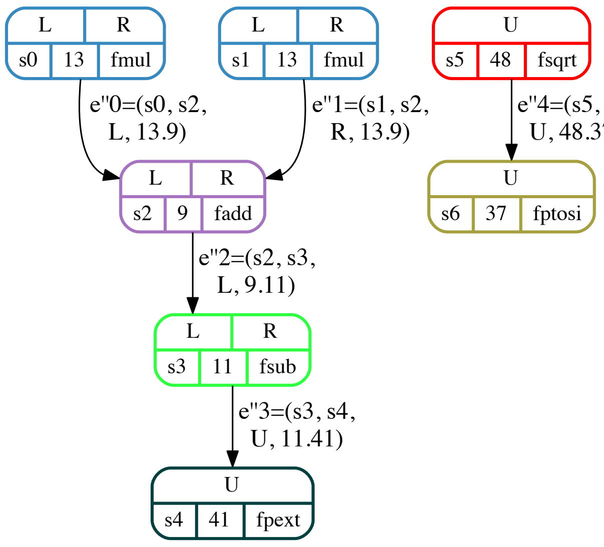
This work proposes a novel Connectivity Graph, which together with the Compatibility Graph enables the enumeration of weakly connected directed subgraphs. Different from the work presented in [45], the Bron-Kerbosch clique enumeration algorithm [46] does not require any modification. The connectivity graph is defined as follows:

Definition 2.11. The connectivity graph $G_w = (V_w, E_w)$, corresponding to a pair of data-flow graphs G_1 and G_2, is a simple undirected graph, where each vertex $v_i \in V_w$ corresponds to an edge mapping. There is an edge $e_i = (u_i, v_i) \in E_w$ *iff* the edge mappings represented by u_i and v_i are adjacent to each other or connected to each other by a path of connected edge mappings in the corresponding graphs G_1 and G_2.

Example 2.12. Let G_1 and G_2 be the data-flow graphs depicted in Fig. 2.4, together with the Connectivity Graph G_w depicted in Fig. 2.7(a). It can be observed that the edge mappings e_3/e'_2 and e_5/e'_3 are connected by an edge in G_w, because



(a) Disconnected subgraph 1.



(b) Disconnected subgraph 2.

Figure 2.6: Disconnected subgraphs extracted from the compatibility graph shown in Fig. 2.5(a). The first is a disconnected subgraph based on the clique presented in Fig. 2.5(b), while the second is a disconnected subgraph based on the maximum clique presented in Fig. 2.5(c). For the sake of simplicity, the input operands are not shown.

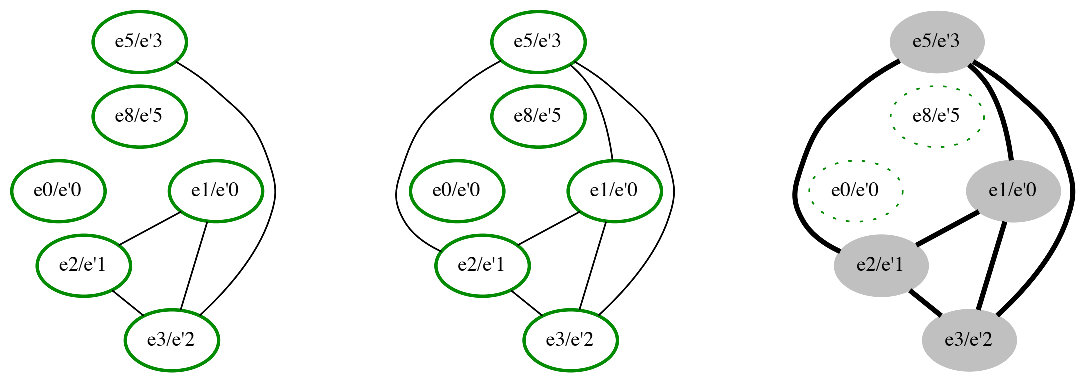
they are adjacent to each other in Fig. 2.4. In other words, edges e_3 and e_5 of G_1 share vertex a_5, while edges e'_2 and e'_3 of G_2 share vertex b_3, what makes both edge mappings adjacent to each other as well. Moreover, it can be noted that the edge mappings e_2/e'_1 and e_5/e'_3 are not connected by an edge in G_w, although they are connected to each other through the edge mapping e_3/e'_2. Thus, they are connected to each other, as depicted in Fig. 2.7(b).

The connectivity graph G_w and its aspects regarding adjacent and connected edges are formally explained and characterized below. First, let's formalize the edge mappings that are adjacent to each other:

Lemma 2.13. *Let G_1 and G_2 be two data-flow graphs. Two edge mappings e_i/e'_j and e_f/e'_k are connected in the connectivity graph G_w whenever the end-points of edges e_i and e_f coincide in G_1 and the same end-points of edges e_f and e'_k coincide in G_2.*

Proof. Let e_i/e'_j and e_f/e'_k be two distinct edge mappings, such that $e_i = (u_i, v_i)$, $e'_j = (u'_j, v'_j)$, $e_f = (u_f, v_f)$ and $e'_k = (u'_k, v'_k)$. There are four possible situations such that the end-points of two edges can coincide:

1. Two edge mappings are adjacent in G_1 and in G_2, respectively, if $u_i = u_f$, both of G_1, and $u'_j = u'_k$, both of G_2;



(a) Connectivity Graph with adjacent edge mappings only.

(b) Connectivity Graph with adjacent and connected edge mappings.

(c) Maximum Clique.

Figure 2.7: Connectivity Graphs generated from the edge mappings shown in Fig. 2.4, together with its maximum clique. The first connectivity graph represent only the edge mappings that are adjacent to each other, while the second connectivity graph includes the edge mappings that are connected to each other via another edge mapping.

2. Two edge mappings are adjacent in G_1 and in G_2, respectively, if $v_i = v_f$, both of G_1, and $v'_j = v'_k$, both of G_2;
3. Two edge mappings are adjacent in G_1 and in G_2, respectively, if $u_i = v_f$, both of G_1, and $u'_j = v'_k$, both of G_2;
4. Two edge mappings are adjacent in G_1 and in G_2, respectively, if $v_i = u_f$, both of G_1, and $v'_j = u'_k$, both of G_2.

□

Example 2.14. In Fig.2.4, edges e_1 and e_3 of G_1 are compatible to edges e'_0 and e'_2 of G_2, respectively. Thus, $e_1 = (a_1, a_4)$ can be mapped to $e'_0 = (b_0, b_2)$, while $e_3 = (a_4, a_5)$ can be mapped to $e'_2 = (b_2, b_3)$. Furthermore, these two edge mappings (e_1/e'_0 and e_3/e'_2) are connected to each other by vertex a_4 in G_1 and vertex b_2 in G_2. Thus, there is a corresponding edge in the connectivity graph connecting these two edge mappings. On the other hand, edges e_1 and e_8 of G_1 and edges e'_0 and e'_5 of G_2 are not connected in the connectivity graph, even though they are compatible. Thats because there are no other connected edge mappings in the path between edges e_1 and e_8 of G_1 and between edges e'_0 and e'_5 of G_2. Fig. 2.7(a) shows an example of connectivity graph which obeys to Lemma 2.13.

Lemma 2.15. *Let G_1 and G_2 be two data-flow graphs. Also, let e_i/e'_j and e_f/e'_k be two distinct edge mappings that are not yet connected in the connectivity graph G_w. If these two edges mappings are reachable by a path $P \in G_w$, then they are also connected to each other by an edge in G_w.*

Proof. Let e_i/e'_j and e_f/e'_k be two distinct edge mappings, such that e_i/e'_j is not yet connected to e_f/e'_k by an edge in the connectivity graph G_w, which means that the edges being mapped in G_1 and G_2 do not share a vertex, *i.e.*, they are not adjacent. Let P be a path in G_w connecting e_i/e'_j and e_f/e'_k, which means that these edge mappings are reachable via other edge mappings in path P. Therefore, by transitivity, if e_f/e'_k is reachable by e_f/e'_k via a path of edge mappings, then e_i/e'_j is connected to e_f/e'_k by an edge in G_w. □

Example 2.16. In Fig. 2.7(a), the edge mapping e_5/e'_3 is only connected to the edge mapping e_3/e'_2, suggesting that, in Fig. 2.4, edge e_5 is only connected to edge e_3, both of G_1, and edge e'_3 is only connected to edge e'_2, both of G_2. It is obvious that this is not true, because edge e_5 is indirectly connected to edges e_1 and e_2, regarded to G_1, and edge e'_3 is indirectly connected to edges e'_0 and e'_1, regarded to G_2. Therefore, by transitivity, the edge mapping e_5/e'_3 can also be connected by an edge to the edge mappings e_1/e'_0 and e_2/e'_1. Fig. 2.7(b) shows an example of connectivity graph which obeys to Lemma 2.15.

2.2.1.4 Intersection

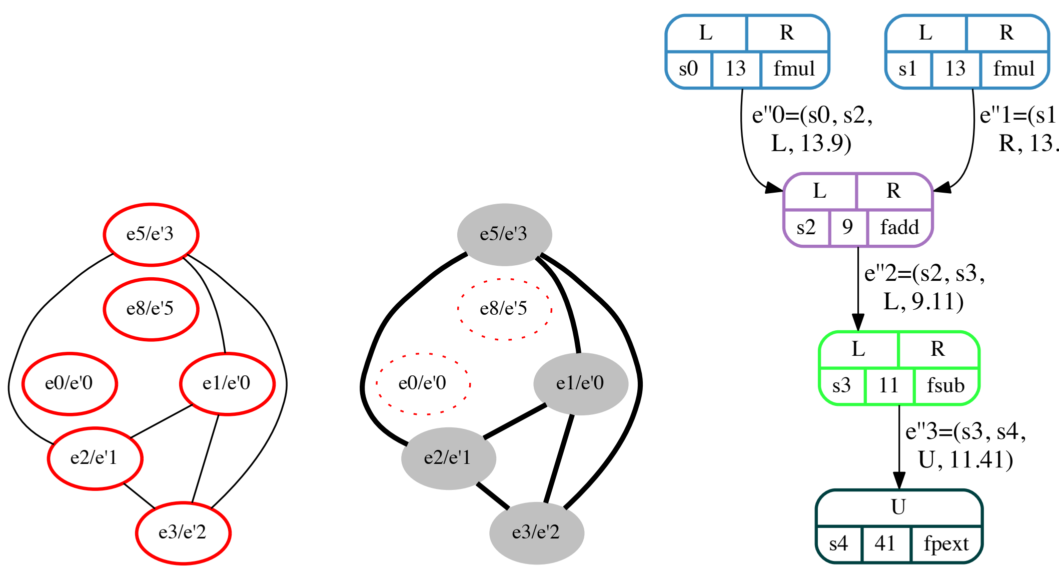
In this Section, the intersection of the compatibility and connectivity graphs is described. The intersection is formalized as follows:

Theorem 2.17. *The intersection of the compatibility graph G_c and the connectivity graph G_w, corresponding to a pair of data-flow graphs G_1 and G_2, results in a new graph G_k that contains only edges present both in the compatibility graph and in the connectivity graph. Thus, G_k contains edge mappings that are both compatible and connected.*

Proof. Let G_c and G_w be a compatibility and a connectivity graph, respectively, corresponding to a pair of data-flow graphs G_1 and G_2. The intersection $G_c \cap G_w$ produces a graph G_k which contains only edges present both in the first and the second graphs. Therefore, these remaining edges correspond to edge mappings that are compatible and connected at the same time. □

Example 2.18. Fig. 2.8(a) exemplifies the intersected graph of the compatibility graph shown in Fig. 2.5(a) with the connectivity graph shown in Fig. 2.7(a), which is coincidently equal to the intersected graph. The enumerated maximum clique shown in Fig. 2.8(b) maps to the connected subgraph shown in Fig. 2.8(c).

Enumerating maximum cliques from the graph G_k significantly helps to reduce the search space, resulting only in weakly connected directed subgraphs. The enumeration of maximum cliques from the compatibility graph G_c was often not feasible especially due to the high memory consumption of the clique enumeration



(a) Intersected Graph. (b) Maximum Clique. (c) Connected Subgraph.

Figure 2.8: Intersected Graph G_k generated from the compatibility graph G_c depicted in Fig. 2.5(a) and from the connectivity graph depicted in Fig. 2.7(a), accompanied by its maximum clique and corresponding weakly connected directed subgraph.

algorithm. Nevertheless, the proposed tool allows the user to decide whether to enumerate cliques from the graph G_k or directly from the compatibility graph G_c, which produces disconnected subgraphs. Moreover, the compatibility graph presented in [28] also includes information about possible mappings of vertices between a pair of DFGs, in order to maximize the datapath merging solution. However, for the CI identification problem, custom instructions have at least two basic operations. Thus, including information about mapping of vertices only contributes to increase the size of the compatibility graph and the clique-enumeration execution time.

2.2.2 Commutativity and associativity

The exploitation of commutativity and associativity properties of operations during the CI identification methodology are novel contributions of this work. Even though the exploitation of commutativity property has been proposed in the hardware sharing domain [19, 28], it is used in this thesis as basis for the associative property. Also, the produced custom instructions have their commutative counterparts generated in order to group the custom instructions which have the same expression into the same library.

The commutativity property derives from a relaxation of the rules applied for edge mapping. The second derives from the commutativity property itself, with restrictions regarding floating-point operations, which are not associative [47]. Distributivity is not yet supported and is proposed as a future work of this thesis. A list of considered operations and their properties is shown in Table 2.1.

Table 2.1: Binary LLVM-IR operations.

Operation	Commutative	Associative
add	Yes	Yes
sub	No	No
div	No	No
mul	Yes	Yes
and	Yes	Yes
shl	No	No
ashr	No	No
or	Yes	Yes
xor	Yes	Yes
fadd	Yes	No
fmul	Yes	No
fdiv	No	No

A binary operation is said to be commutative *iff* the order of its operands does not affect the final result, *i.e.*, for a set S of operands x and y, a binary operation *op* is said to be commutative *iff* $\forall x, y \in S$:

$$x \; op \; y = y \; op \; x$$

In the CI identification methodology context presented in this work, the commutativity derives from the edge mapping definition with a minor modification: the edge port attribute information is irrelevant *iff* the target vertex is a binary commutative operation. In this case, the edge mapping is called *relaxed edge mapping*, because the edge port attribute is ignored during the edge mapping process and, thus, only the edge color is taken into account. This allows any edge to be mapped to any other edge of the same color f, regardless of their port information (L,R or U).

Fig. 2.9 depicts a pair of data-flow graphs G_1 and G_2, their edge mappings and *relaxed edge mappings*. Observe that operations a_4 of G_1 and b_4 of G_2 are binary non-commutative operations, which means that their input ports cannot be switched. Therefore, edges e_1 and e_2 of G_1 can only be mapped to edges e'_2 and e'_3 of G_2, respectively. On the other hand, observe that operations a_5 of G_1 and b_5 of G_2 are binary commutative operations. Thus, even though edge e_3 arrives at a port in G_1 that is different from the port that edge e'_4 arrives at G_2, they can still be mapped because of the commutative property of their target vertex and the relaxation of their edge mappings. Otherwise, only the edge mappings e_1/e'_2 and e_2/e'_3 would be allowed, resulting in a much smaller subgraph.

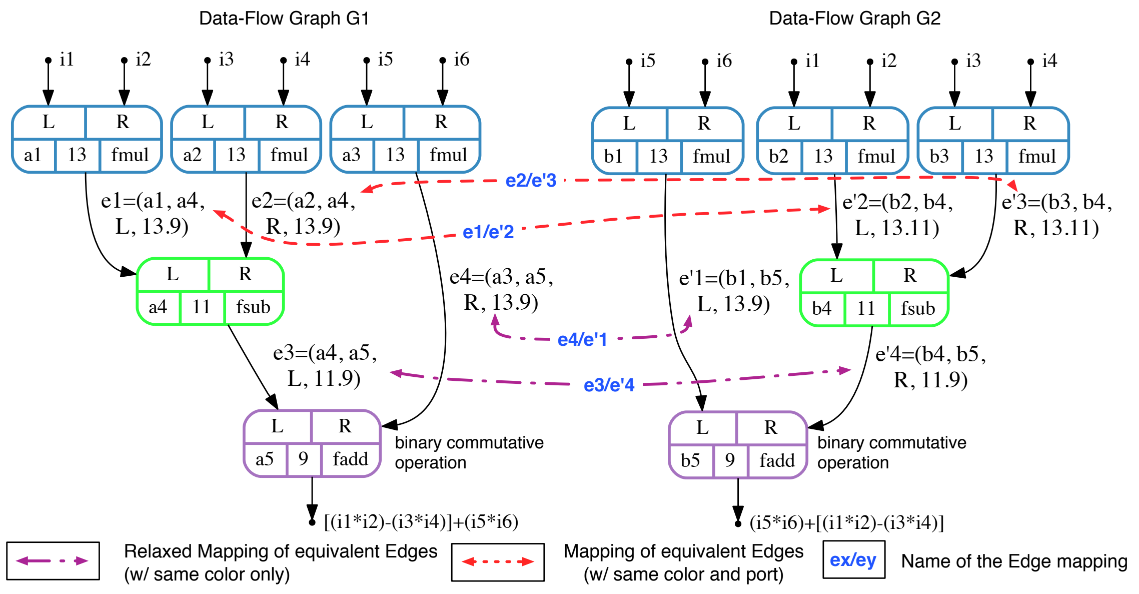


Figure 2.9: A pair of Data-Flow Graphs and their *relaxed mapping* of equivalent edges, which are represented by pink dashed arrows and labeled according to the given edges id. For instance, edge $e_3 = (a_4, a_5, L, 11.9) \in E_1$ can be mapped to edge $e'_4 = (b_4, b_5, R, 11.9) \in E_2$, because they have the same color $f = 11.9$. The port information is ignored for edges with a target vertex that is binary and commutative. Thanks to the commutative property, graphs G_1 and G_2 can be completely mapped to each other. Otherwise, only a subset of edges would be mapped, producing a smaller common subgraph.

A binary operation is said to be associative for a set S of operands x, y and z *iff* $\forall x, y, z \in S$:

$$(x\ op\ y)\ op\ z = x\ op\ (y\ op\ z)$$

In the CI identification methodology, the associativity derives from the *relaxed edge mapping* commutative definition, excluding floating-point operations. Also, it requires that all the operations of the graph are the same. Fig. 2.10 depicts a pair of data-flow graphs G_1 and G_2, and their *relaxed edge mappings*. Observe that all operations are binary associative operations. Therefore, edges e_1 and e_2 of G_1 can be mapped to edges e'_1 and e'_2 of G_2, in any order. The corresponding compatibility graph should identify e_1/e'_1 and e_2/e'_2 as one group of compatible mappings and e_1/e'_2 and e_2/e'_1 as another group of compatible mappings, because any other combination is incompatible. In both cases, the resulting subgraph consists of three multiplication operations connected one after the other, with its input operands entering at any port (left or right).

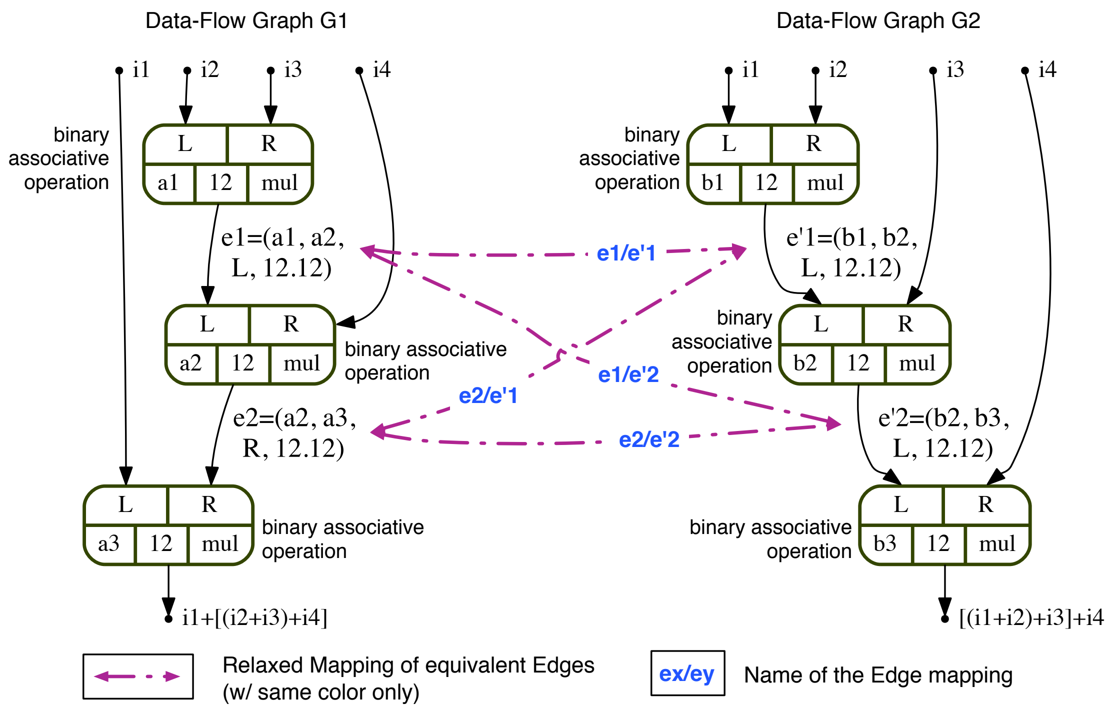


Figure 2.10: A pair of Data-Flow Graphs and their *relaxed mapping* of equivalent edges, which are represented by pink dashed arrows and labeled according to the given edges id. For instance, edge $e_2 = (a_2, a_3, R, 12.12) \in G_1$ can be mapped to edge $e'_2 = (b_2, b_3, L, 12.12) \in G_2$, because they have the same color $f = 12.12$. The port information is ignored for edges with a target vertex that is binary and associative. The commutative and associative properties enable the mapping of graphs G_1 onto G_2, or vice-versa.

2.2.3 The algorithm

In this Section, the custom instruction identification algorithm is subdivided into Algorithm 1 and Algorithm 2. Together, they can automatically identify common frequently executed patterns of an application, *i.e.*, the Maximum Common Subgraphs that exist among the data-flow graph representation of the most frequently executed basic blocks of an application. Therefore, a set of data-flow graphs $G = \{G_1, G_2, \cdots, G_n\}$ is required, each one extracted from a basic block of the application that is being analyzed. The result is a database of custom instructions $P = \{p_1, p_2, \cdots, p_n\}$, *i.e.*, common patterns that are identified during the subgraph isomorphism comparison of each pair of data-flow graphs, G_i and G_j, taken from the set G.

The subgraph enumeration function, shown in Algorithm 2, is a part of the whole algorithm that involved most of the work described in Section 2.2. The function is responsible for: (i) mapping each edge of G_i to the equivalent edge of G_j, (ii) checking which mappings are compatible to each other, (iii) checking which mappings are adjacent and connected to each other, (iv) enumerating all the

Algorithm 1 CI Identification

```
Require: application bb. data-flow graphs $G = \{G_1, G_2, \cdots, G_n\}$
Ensure: database of custom instructions $P = \{p_1, p_2, \cdots, p_m\}$
 1: for $i \leftarrow 0$ to $|G| - 1$ do
 2:     for $j \leftarrow i + 1$ to $|G|$ do
 3:         S ← EnumerateAllCommonSubgraphs($G_i$,$G_j$);
 4:         if $S \neq \emptyset$ then
 5:             for $t \leftarrow 0$ to $|S|$ do
 6:                 if $S_t \notin P$ then
 7:                     $C \leftarrow$ generateCommutativePatterns($S_t$);
 8:                     $A \leftarrow$ generateAssociativePatterns($S_t$);
 9:                     if $((C \cap P) \neq \emptyset)$ or $((A \cap P) \neq \emptyset)$ then
10:                         hist[$S_t$] ← hist[$S_t$] + frequency[$S_t$];    ▷ pattern already included;
11:                     else
12:                         addToLibrary($P, S_t$);    ▷ $S_t$ does not exists in $P$; include $S_t$ into $P$;
13:                         hist[$S_t$] ← frequency[$S_t$];    ▷ update frequency counter;
14:                 else
15:                     hist[$S_t$] ← hist[$S_t$] + frequency[$S_t$];    ▷ pattern already included;
16: for $i \leftarrow 0$ to $|P|$ do
17:     if $P_i$ is associative then
18:         $H \leftarrow$ generateReAssociations($P_i$);
19:         addToLibrary($R, H$);    ▷ include all re-associations $H$ into $R$;
```

maximum cliques from the intersection graph (or optionally from the compatibility graph) and, finally, (v) producing a set of subgraphs S extracted from each maximum clique.

Once the subgraphs have been enumerated, the program described in Algorithm 1 is responsible for the identification of which subgraphs are already included in the database of custom instructions P. In summary, for each identified subgraph, the algorithm includes it in the database of patterns P if and only if the identified pattern has not yet been included in the database, also taking into consideration all the commutative and associative combinations of the given subgraph, as shown in line 7 and line 8 of Algorithm 1. Thus, if the identified subgraph is already present in the database, or one of its commutative/associative combinations, then only the frequency of execution of the given subgraph is updated, *i.e.*, accumulated. Otherwise, the subgraph is included in the database as a new pattern occurrence.

The commutative and associative combinations are produced based on the commutativity and associativity properties of each LLVM-IR operation that exists in the subgraph. For instance, if there is a commutative operation in subgraph S, then two combinations of this subgraph are created: one with the operands arriving at their original input ports of the commutative operation and the other with the operands of the commutative operation swapped. The same strategy is used for the associative LLVM-IR operations that exists in the subgraph, except that all the operations need to be the same.

Finally, once the library of common patterns P has been produced, the associa-

Algorithm 2 Maximum Common Subgraphs Enumeration

```
function EnumerateAllCommonSubgraphs($G_1$,$G_2$)
    $C = \{c_1, c_2, \cdots, c_n\}$                ▷ set of all cliques enumerated between $G_1$ and $G_2$
    $M = \{m_1, m_2, \cdots, m_n\}$                ▷ set of all edge mappings between $G_1$ and $G_2$
    $G_c = (V_c, E_c), G_w = (V_w, E_w), G_k = (V_k, E_k)$  ▷ compatibility, connectivity and intersection graphs
    $S = \{s_1, s_2, \cdots, s_n\}$                ▷ set of all subgraphs identified between $G_1$ and $G_2$
    for each edge $e_i$ in $G_1$ do
        for each edge $e_j$ in $G_2$ do
            if $edge\_color[e_i] = edge\_color[e_j]$ then
                if $edge\_port[e_i] = edge\_port[e_j]$ then
                    $m_t \leftarrow$ MAP$(e_i, e_j)$;          ▷ map edge $e_i$ to edge $e_j$
                    addMapping$(M, m_t)$;                     ▷ append $m_t$ to $M$
                    addVertex$(G_c, m_t)$;                    ▷ $m_t$ becomes a vertex of $G_c$
                    addVertex$(G_w, m_t)$;                    ▷ $m_t$ becomes a vertex of $G_w$
    for each mapping of edges $m_i$ in $M$ do
        for each mapping of edges $m_j$ in $M$ do
            if $m_i$ is compatible to $m_j$ then
                addEdge$(G_c, m_i, m_j)$
            if $m_i$ is connected to $m_j$ then
                addEdge$(G_c, m_i, m_j)$
    $G_k \leftarrow G_c \cap G_w$;                 ▷ intersect compatibility and connectivity graphs
    if connectivity is enabled then
        $C \leftarrow$ enumerateAllMaximumCliques$(G_k)$;   ▷ Bron-Kerbosch algorithm
    else
        $C \leftarrow$ enumerateAllMaximumCliques$(G_c)$;   ▷ Bron-Kerbosch algorithm
    for each clique $c_i$ in $C$ do
        $G \leftarrow$ extractSubgraph$(c_i)$;
        $S \leftarrow S \cup G$                    ▷ include the subgraph $G$ into $S$
    return $S$;                                    ▷ return the set of subgraphs $S$
```

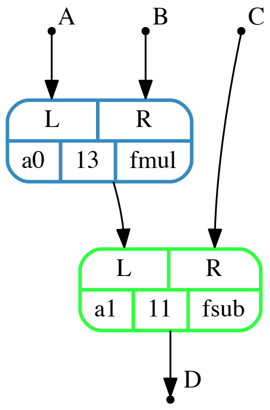
tive patterns are re-associated, as shown in line 18 of Algorithm 1. The re-association produces all the other associative patterns, with the same semantics, and that are potentially more efficient in terms of performance, as will be presented in Section 2.4.2.

2.3 ASIP implementation

The last part of the proposed framework is the ASIP implementation. The commercially available VLIW-based ASIP that is used in this thesis can be tailored in order to better exploit the application Instruction Level Parallelism (ILP). After the identification process, the library of custom instructions can be automatically added to the processor's library of default instructions, together with their corresponding function units (FUs) characterized regarding their physical features (processing speed, circuit area and energy consumption).

2.3.1 Instruction set customization

For the VLIW-ASIP extension, a high level C-like specification language (also referred to as Machine Description Language, or MDL) can be used. It allows the specification of several macro-architectural elements of a VLIW-ASIP, such as Register Files, Buses, Issues-Slots, Memories, Function Units, etc. For instance, the common pattern graph of a 32-bit unsigned multiply-subtraction operation, depicted in Fig. 2.11(a), translates to its corresponding MDL specification shown in Fig. 2.11(b).



(a) Multiply-Subtract Data-Flow Graph.

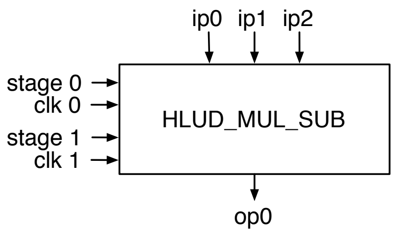
```
OP mul_sub (Unsigned A,B,C)->(Unsigned D)
{
   SEM D(A,B,C) = {
      D = (A * B) - C;
   }
};
```

(b) Multiply-subtract semantics MDL specification.

Figure 2.11: Multiply-subtraction custom instruction data-flow graph together with its semantics specification using the ASIP Machine Description Language (MDL).

The custom instruction function unit corresponding to this graph and its MDL specification is produced based on the operation semantics, using the same MDL specification, requiring only a translation step from the MDL specification to the micro-architecture corresponding specification in a Hardware Description Language (HDL) for later implementation in a custom or semi-custom Integrated Circuit (IC), or Reconfigurable Logic (FPGA). Nevertheless, if the timing information is provided, *i.e.*, the custom instruction time-shape, the retargetable compiler can already map the new custom instructions (CIs) onto their corresponding issue-slots for cycle-accurate simulation results. The function unit timing specifies at which time (clock cycle) each basic operation should occur. The function unit implementing the multiply-subtract custom instruction is depicted in Fig. 2.12(a).

It can be noted from the MDL specification in Fig. 2.12(b) that the function unit time-shape specifies at which clock cycles the function unit ports can be used for sampling an input value or producing an output value. In that case, ports ip_0, ip_1, ip_2 are sampled at cycle 0, while the output port is sampled at cycle 1. Therefore, the function unit is pipelined in two stages and takes a total of 2 cycles to produce the result. If all the ports should be sampled at the same clock cycle, then the



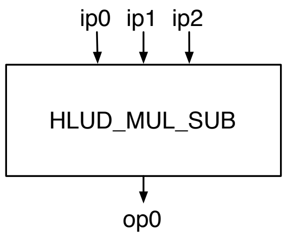
(a) Multiply-subtract function unit.

```
FU HLUD_MUL_SUB
  ( Port32 ip0, ip1, ip2 ) -> ( Port32 op0 )
{
   Cycle[0] := {ip0,ip1,ip2};
   Cycle[1] := {op0};

   msub : op0 = mul_sub(ip0,ip1,ip2);
};
```

(b) Multiply-subtract function unit MDL specification.

Figure 2.12: Multiply-subtraction function unit (HLUD_MUL_SUB) together with its specification using the ASIP Machine Description Language (MDL).

specification is modified as shown in Fig. 2.13(b).



(a) One-cycle multiply-subtract function unit.

```
FU HLUD_MUL_SUB
  ( Port32 ip0, ip1, ip2 ) -> ( Port32 op0 )
{
   Cycle[0] := {ip0,ip1,ip2,op0};

   msub : op0 = mul_sub(ip0,ip1,ip2);
};
```

(b) One-cycle multiply-subtract function unit MDL specification.

Figure 2.13: One-cycle multiply-subtraction function unit (HLUD_MUL_SUB) together with its specification using the ASIP Machine Description Language (MDL).

Based on such time-shape information, the VLIW-ASIP compiler can schedule the new custom instructions together with the regular instructions. Thus, according to the custom instruction time-shape and the application cycle-accurate simulation, the acceleration provided by a given custom instruction can be evaluated. Let's consider, for instance, the kernel code presented in Listing 2.1, which will be executed by the VLIW-ASIP. The kernel code represents a multiply-subtraction operation on an element-by-element basis of vectors A,B and C, with 100 elements each.

Listing 2.1: Vector multiply-subtraction operation.

```c
int A[100];
int B[100];
int C[100];
int D[100];

void kernel(void)
{
  int i;
  for(i = 0 ; i < 100 ; i++)
    D[i] = A[i] * B[i] - C[i];
}
```

Without the multiply-subtract custom instruction, the scheduler would have to emit a multiplication operation before a subtraction operation, inside each iteration of the *for loop*. If the multiplication and the subtraction requires one cycle each to execute, then two cycles would be necessary to compute the required result at each iteration. On the other hand, using the multiply-subtract custom instruction presented in Fig. 2.13, with a time-shape of one cycle, both multiplication and subtraction operations could be executed in one cycle, at each iteration. Hence, one cycle would be saved at each iteration of the *loop*, as shown in Listing 2.2.

Listing 2.2: Vector multiply-subtraction custom operation.

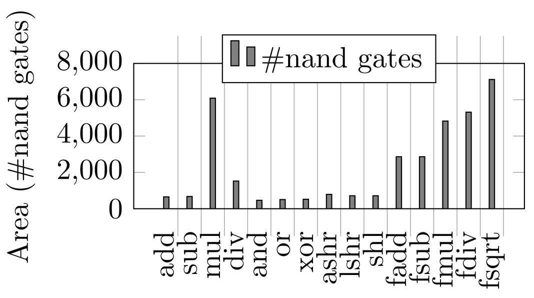
```
int A[100];
int B[100];
int C[100];
int D[100];

void kernel(void)
{
  int i;
  for(i = 0 ; i < 100 ; i++)
    D[i] = OP_MUL_SUB(A[i],B[i],C[i]);
}
```

2.3.2 Characterizing circuit area and energy

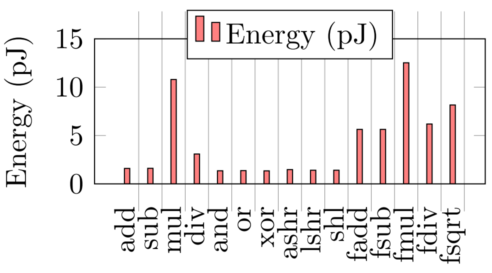
The circuit area and energy consumption of each custom instruction is characterized from the individual circuit area and energy consumption synthesis results of each of its basic operators, *i.e.*, function units. Thus, the synthesis tool takes a RTL hardware description in VHDL of each basic operator and produces reports on their circuit area and energy consumption.

Synthesis results of the main LLVM-IR basic operations are presented in Fig. 2.14(a) and Fig. 2.14(b), while further details are shown in Table 2.2. These are the operations that are enabled by default in the proposed tool to identify groups of basic operations that, together, can form custom instructions that are executable by the VLIW-ASIP. For instance, memory and branch operations are not supported by the tool, because the underlying VLIW-ASIP does not support the customization of function units that use memory or branch operations. Hence, such operations are disabled by default in the proposed tool and, unfortunately, limits the potential of speedup that could be achieved otherwise. Because of that, some custom instructions will not be identified, such as address calculation of memory operations or custom instructions which ends with a conditional branch operation, both very common in many application domains.

The logic synthesis results of each basic operation are used to estimate the circuit area requirements and energy consumption of the custom instructions. The synthesis results are based on Cadence RTL compiler logic synthesis for a 65nm technology TSMC cell library, set at 5ns timing constraint and applying the compiler's default



(a) LLVM-IR circuit area estimation.



(b) LLVM-IR Energy estimation.

Figure 2.14: Circuit area and energy estimation of the main LLVM-IR basic operations. Each basic operation of the LLVM-IR was separately described in VHDL and synthesized using Cadence RTL-Compiler, based on TSMC 65nm technology, set at 5ns timing constraint and applying default switching activities.

transistor switching activities. The latency of all basic operations is of one execution cycle, with exception of the division and square-root operations, whether in floating-point or integer arithmetic. These are set to eight execution cycles.

Table 2.2: Circuit area (in number of *nand* gates) and energy consumption (pJ) of the main 32-bit LLVM-IR basic operations, based on TSMC 65nm technology.

Operation	Type	Semantics	Area	Energy
add	Unsigned	$R = A + B$	651	1.5978
sub	Unsigned	$R = A - B$	673	1.6083
mul	Unsigned	$R = A * B$	6087	10.7941
div	Unsigned	$R = A/B$	1523	3.0878
and	Unsigned	$R = A\&B$	464	1.3543
or	Unsigned	$R = A\|B$	504	1.3715
xor	Unsigned	R = AˆB	520	1.3460
ashr	Unsigned	$R = A >> B$	791	1.4756
lshr	Unsigned	$R = A >> B$	713	1.4100
shl	Unsigned	$R = A << B$	713	1.4103
fpadd	Float	$R = A + B$	2862	5.6334
fpsub	Float	$R = A - B$	2862	5.6334
fpmul	Float	$R = A * B$	4827	12.5211
fpdiv	Float	$R = A/B$	5317	6.1898
fpsqrt	Float	$R = sqrt(A)$	7115	8.1580

The circuit area of each custom instruction is estimated based on the sum of the circuit area of each of its basic operations. Therefore, the data-flow graph $G = (V, E)$ of a custom instruction, where each vertex $v \in V = \{v_1, v_2, \cdots, v_n\}$ corresponds to a basic operation, has a total approximate circuit area as described in Equation 2.1.

$$TotalArea = \sum_{i=0}^{n} Area(v_i) \tag{2.1}$$

The energy consumption of each custom instruction is also estimated based on the sum of the energy consumption of each of its basic operations. Therefore, the custom instruction has a total approximate energy consumption as described in Equation 2.2.

$$TotalEnergy = \sum_{i=0}^{n} Energy(v_i) \tag{2.2}$$

Consider, for instance, the *dot product* custom instruction data-flow graph depicted in Fig. 2.15, where each vertex $v \in V$, presents its circuit area A and energy consumption E, based on the logic synthesis results of the corresponding basic operators described in Table 2.2. Therefore, it is possible to estimate that the *dot product* custom instruction has a total circuit area of 20205 *nand gates* and consumes a total energy of 48.83 pJ.

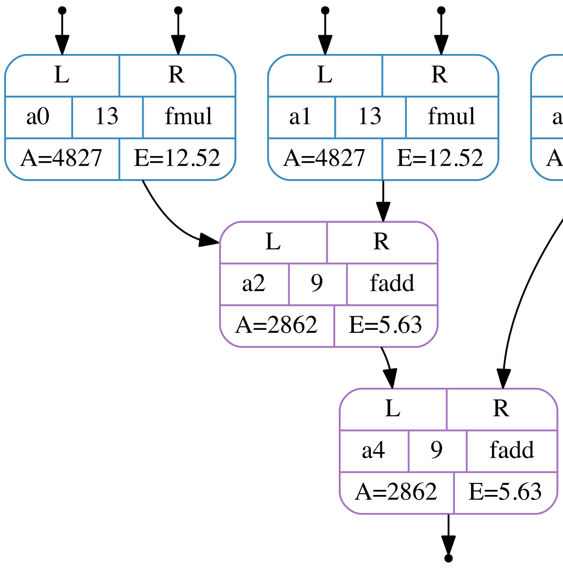
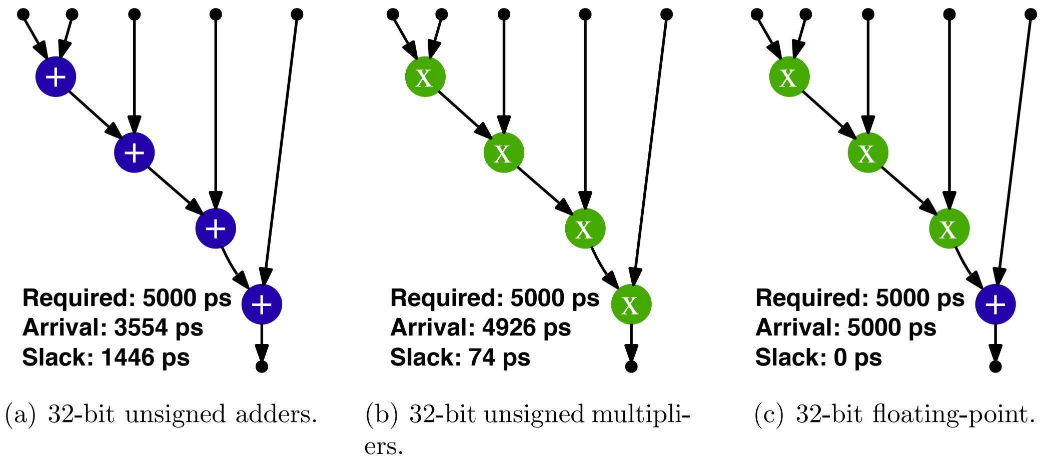


Figure 2.15: Dot product custom instruction data-flow graph.

2.3.3 Custom instructions time-shape

A subset of the most timing critical custom instructions were synthesized with a 5ns timing constraint, using TSMC 65nm technology, in order to estimate whether the custom instructions can operate in one execution cycle, as shown in Fig. 2.16. It can be noted that for the three custom instructions, their timing slack remained within the 5ns timing constraint and, thus, may operate in one execution cycle. However, in order to obtain precise information regarding the delay of the custom instructions, the whole VLIW-ASIP should be synthesized together with the custom function units of the custom instructions. Unfortunately, the VLIW-ASIP and the custom function units could not be synthesized, due to the high memory consumption for the synthesis of the VLIW-ASIP. Thus, all the identified custom instructions were set to operate in one execution cycle for the cycle-accurate simulation results presented in Section 2.4.



(a) 32-bit unsigned adders. (b) 32-bit unsigned multipliers. (c) 32-bit floating-point.

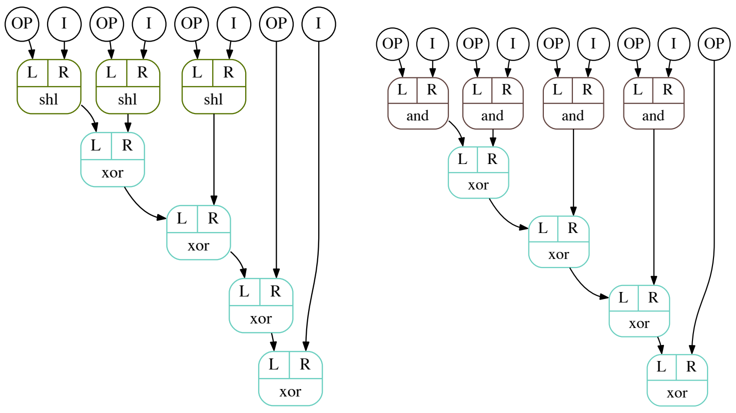
Figure 2.16: High-latency custom instructions.

2.4 Experimental results

In this Section the experimental results regarding instruction set customization are discussed for the set of benchmark applications presented in Section 1.4. The custom instruction identification framework applied to the set of benchmark applications produced a library of 70 custom instructions in total.

2.4.1 Experimental setup

For each benchmark application one VLIW-ASIP processor was configured with a pair of regular issue-slots (IS) and up to three augmented issue-slots (AIS). The regular issue-slots are mainly responsible for handling memory access and branch operations, besides handling other regular operations. The first augmented issue-slot is one of the regular issue-slots containing a mixture of regular function units and custom function units, while the other two are dedicated issue-slots containing custom function units only. This configuration is used to evaluate how much speedup is obtained from the custom instructions, 1-AIS configuration, and how much is gained from Instruction Level Parallelism (ILP), e.g., 2/3-AIS configurations. Each VLIW-based ASIP was simulated using its own cycle-accurate simulation tool-chain. Circuit area and energy consumption estimates are provided for each custom instruction, based on TSMC 65nm technology with 5ns timing constraint, as presented in Section 2.3.2. The identified library of custom instructions is presented in Appendix A.



(a) AES custom instruction 1. (b) AES custom instruction 2.

Figure 2.17: Example of coarse-grain custom instructions of the AES application, each requiring 4 immediate (I) operands.

2.4.2 Commutative and associative analysis

For each benchmark application a substantial number of basic blocks was extracted from the profiling stage of the custom instruction identification framework, as can be observed in Table 2.3. Also, it can be noted that around 20 patterns were identified for each application based on the data-flow graph comparison performed by the custom instruction identification stage. However, only a subset of the identified patterns were selected for implementation as custom instructions due to limitations of the VLIW-ASIP retargetable compiler and architecture. For instance, many patterns could not be scheduled by the compiler, mostly due to the limitation on the number of input operands that the VLIW-ASIP architecture imposes (only 1 immediate operand is supported). For instance, some of the patterns identified for the AES application makes extensive use of immediate operands, as shown in Fig. 2.17.

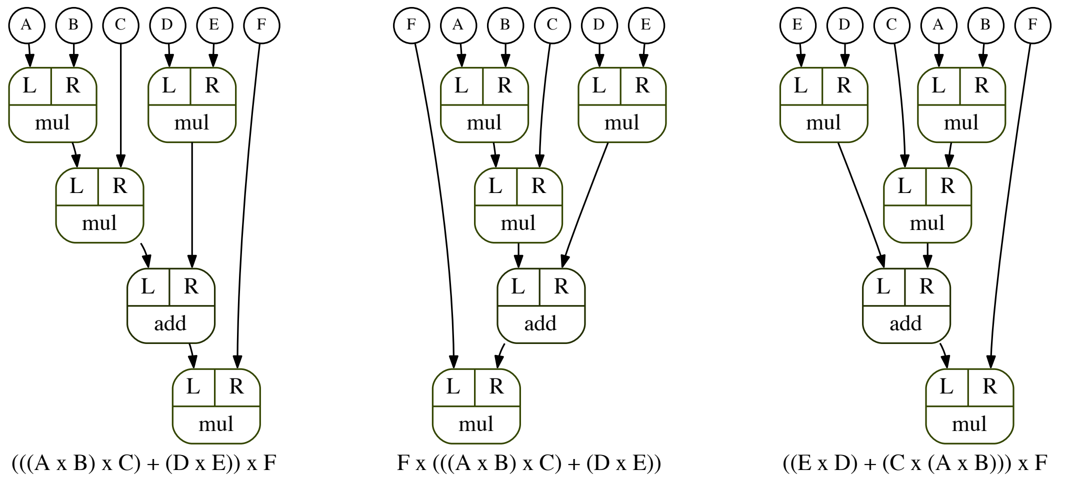
For that reason, fewer patterns were implemented as custom function units into the augmented issue-slots for the AES application, as well as for other applications that require more than one immediate operand. Furthermore, coarse-grain patterns could not be often scheduled too, mainly because of their large number of input operands, which imposes a higher pressure on the register files and makes it more difficult for the compiler to schedule them.

Commutative patterns were often identified, especially for the JPEG application, as can be observed in Table 2.3. All the identified custom instructions are grouped together as a single custom instruction and their execution frequencies are also accumulated. Examples of commutative custom instructions is shown in Fig. 2.18.

Associative patterns, on the other hand, occur less frequently when compared to commutative patterns. In general, only a few associative patterns were identified for

Table 2.3: The identified commutative and (re)-associative patterns

	Basic Block DFGs	Identified Patterns	Commutative Patterns	Associative Pattern	Re-associations	Implemented CIs
AES	24	17	0	3	33	9
SHA	29	10	2	4	17	7
JPEG	108	37	12	2	4	11
MJPEG	92	23	9	1	2	5
CRC	98	2	0	0	0	1
EDGE	83	14	4	0	0	3
RT	79	21	1	1	2	14
VRC	66	13	3	1	2	7
MPSO	56	26	7	1	2	11


A
B
C
D
E
F
L R mul
L R add
(((A x B) x C) + (D x E)) x F
F x (((A x B) x C) + (D x E))
((E x D) + (C x (A x B))) x F

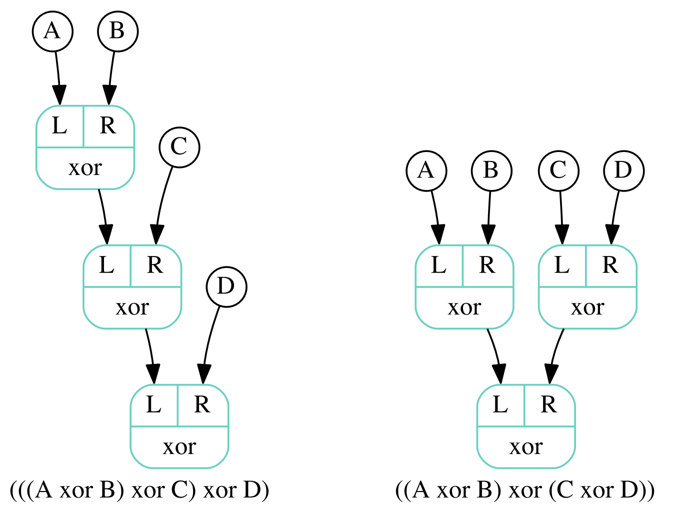
(a) Commutative pattern 1. (b) Commutative pattern 2. (c) Commutative pattern 3.

Figure 2.18: Example of identified commutative custom instructions of the JPEG application.

the set of benchmark applications, with a couple more identified for the AES and SHA applications. These two applications, in particular, consists of several processing steps, each containing sets of binary commutative and associative operations, such as *and*, *xor*, *add*, etc. Therefore, some of the identified patterns consists of a group of only one kind of associative operation and, thus, is also associative, as shown in Fig. 2.19(a). Moreover, the associative patterns can be re-associated in order to produce more efficient patterns, i.e., custom instructions with operations that can be processed in parallel. For instance, in Fig. 2.19(b), it can be noted that a pair of *xor* operations can be processed in parallel, contributing also to reduce the delay of the custom instruction datapath.

2.4.3 Performance, area and energy estimation results

The speedup in comparison to the default VLIW-ASIP instruction set implementation, *i.e.*, without instruction set extensions, is presented in Fig. 2.20, when only one augmented issue slot (1-AIS) is enabled. Further details are presented in Table 2.4, including the number of identified custom instructions (#CIs) for each application



(a) Associative pattern 1. (b) Re-association of pattern 1.

Figure 2.19: Example of identified associative custom instructions of the AES and SHA applications.

and their frequency of access (%Access).

It is possible to observe that when the custom instructions are enabled using 1-AIS, the speedup is substantial and goes up to 25% for the RT application, using the 14 custom instructions that the tool automatically identified for it. Other applications, such as the CRC, achieved a speedup of 14% using only the two custom instructions that the tool identified for it. On average, using 1-AIS configuration and a dozen identified custom instructions provided around 12% of speedup. As expected, most of the custom instructions identified for the RT benchmark are popular operations that are often encountered in the multimedia domain, such as the *dot product* and the *multiply-accumulate* operations. For instance, they were also identified in the VRC application.

Fig. 2.21 depicts how often the custom instructions are accessed (*i.e.*, used) dur-

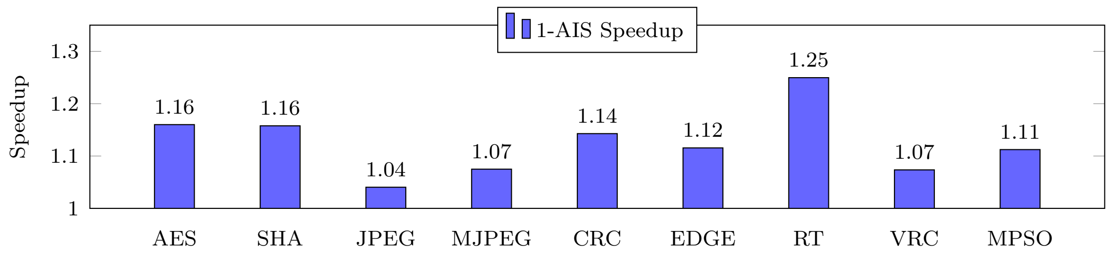


Figure 2.20: Speedup for each benchmark application and its corresponding set of automatically identified custom instructions (CIs), using only one augmented issue-slot (AIS). The speedup is in comparison to using the VLIW-ASIP standard instruction set.

Table 2.4: Speedup for each benchmark and its custom instructions (CIs), together with the number of identified custom instructions (#CIs), their circuit area (in number of nand gates), average (Avg.E.) and total (Total.E.) energy consumption results for one augmented issue-slot (1-AIS).

	CI Speedup			CIs (1-AIS, TSMC 65nm)				
App.	1-AIS	2-AIS	3-AIS	#CIs	%Access	Area #gates	Avg.E. (pJ)	Total E. (nJ)
AES	1.16	1.34	1.34	9	17.30	16285	5.44	16939.66
SHA	1.15	1.20	1.26	7	13.08	12129	5.03	78056.44
JPEG	1.04	1.14	1.14	11	15.30	82682	18.73	156445.40
MJPEG	1.13	1.14	1.15	5	7.37	19535	7.41	24490.13
CRC	1.14	1.14	1.14	1	18.18	984	2.7	143700.48
EDGE	1.12	1.12	1.12	3	2.49	21551	21.48	14004.76
RT	1.25	1.50	1.54	14	8.12	240612	46.06	1229327.13
VRC	1.07	1.19	1.19	7	8.19	78213	30.32	6830686.99
MPSO	1.11	1.11	1.11	13	4.20	111626	23.36	580289.58

ing each application execution time in comparison to the VLIW's default instruction set, for one augmented issue-slot (1-AIS). For instance, the custom instructions that were automatically identified for the RT application were accessed 8.12% of the application's total execution time, leaving the rest for the default instruction set to execute. The custom instructions of some applications were accessed more often, such as the CRC custom instructions, which were accessed 18.18% of the application's total execution time.

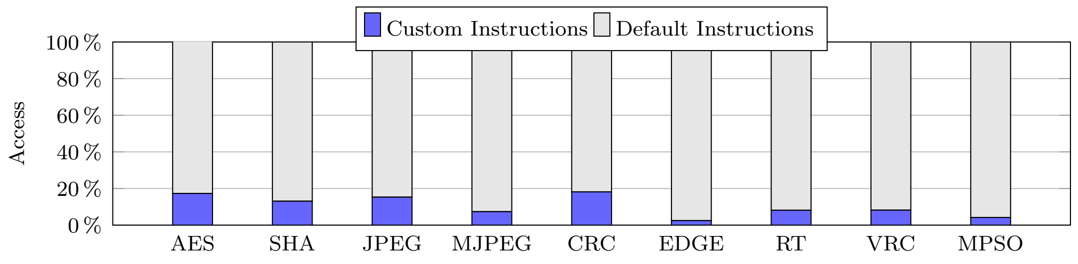


Figure 2.21: Access of Custom Instructions function units compared to the access of Default Instructions function units.

Despite the low access rate of the custom instructions identified for the RT application, in comparison to the other applications, observe that its speedup is the highest amongst all benchmarks. This translates into a good custom instruction efficiency ratio between speedup and custom instruction access rate. In other words, with fewer accesses to its custom instructions, the RT application obtained a better speedup than most of the other applications with twice or more accesses to their respective custom instructions. The custom instructions of the MJPEG, EDGE, VRC and MPSO benchmarks also have a good efficiency ratio, because they were accessed fewer times in comparison to the custom instructions of the AES, SHA, JPEG and CRC benchmarks, and still obtained similar speedup results. The custom instructions of the JPEG application have the worse efficiency ratio, because they produced the lowest speedup and were frequently accessed.

Increasing the number of augmented issue-slots contributes to raise the speedup of the custom instructions by exploiting the ILP that may exist in the application, enabling two or more instructions to be scheduled in parallel by the VLIW compiler. Besides, it helps to reduce the pressure on a single register file (1-AIS), spreading the data over two or more register files, which can be accessed from all issue-slots. In this way, custom instructions can be scheduled more frequently, because data might be available immediately throughout several register files. Thus, the speedup of most applications raised substantially when 2 augmented issue-slots were enabled, because more custom instructions were scheduled in parallel or within the same issue-slot, as depicted in Fig. 2.22. The speedup of the CRC, EDGE and MPSO benchmarks did not change significantly when 2 or more augmented issue-slots were enabled and, thus, are not present in Fig. 2.22.

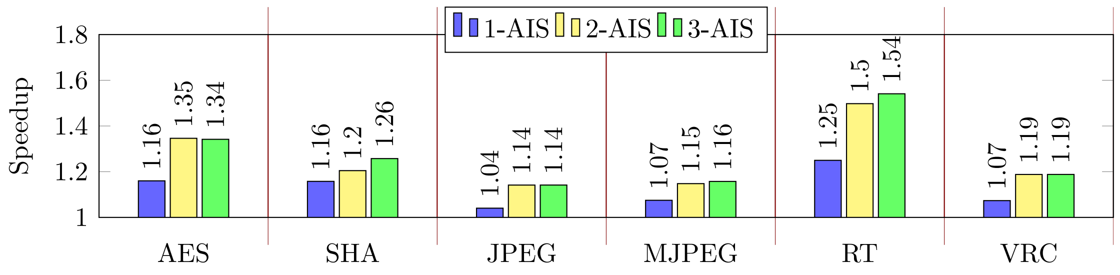
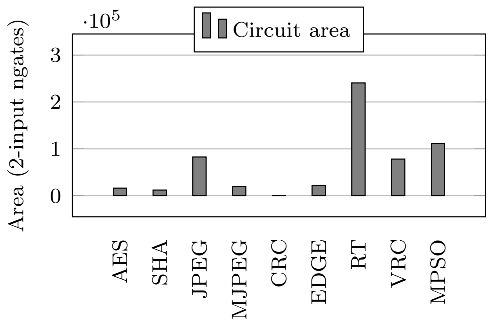
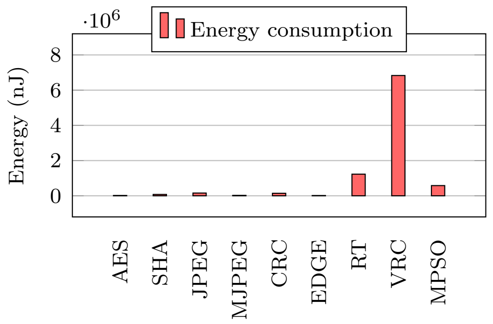


Figure 2.22: Subset of benchmark applications that obtained speedup when varying the number of augmented issue-slots. The speedup is in comparison to using the VLIW-ASIP standard instruction set.

When 3 augmented issue-slots were enabled, the speedup slightly improved for applications such as SHA, MJPEG and RT. For other applications, such as JPEG and VRC, using a 3-AIS configuration did not have any significant impact on the application's performance. In the worst case, an application's performance can also slightly deteriorate, as for the AES application. A careful analysis of the instruction scheduling of each application revealed few opportunities for additional ILP exploitation as more augmented issue slots are included and used, which can be the cause for the lack of improvement. Also, the additional issue-slots are often wasted during most cycles and are only used for the sporadic execution of custom instructions. Furthermore, the issue-slot that handles memory load/store operations is often accessed. Some custom instructions may require several input operands to be available at the moment of its execution. Thus, if such data is not available, the custom instruction may be delayed or not executed at all, prioritizing the usage of the default instruction set. Splitting the required data among several different memories may allow two or more memory operations two be scheduled in parallel by two or more issue-slots. Consequently, data can be made available more frequently,



(a) Total circuit area estimation.



(b) Total energy consumption estimation.

Figure 2.23: Circuit area and total energy consumption estimations for each benchmark application and its corresponding set of automatically identified custom instructions, for only one augmented issue-slot.

allowing the custom instructions to be scheduled.

Circuit area and energy consumption estimative are given in Fig. 2.23, while specific details can be found in Table 2.4. Replicating issue-slots to exploit ILP is a common practice, but it will also demand more circuit area and consume more energy. Thus, there is a trade-off between performance, circuit area and energy. Including a second augmented issue-slot only makes sense if the additional speedup compensates the higher circuit area requirement and energy consumption. From these results it is possible to observe that the area and energy consumption for the custom instructions of the RT, VRC and MPSO applications is considerably higher than those of the other applications. One reason for such high area and energy consumption arises from the fact that these three applications make extensive use of floating-point computation, which is known to require more area and consume more energy. Another reason is the larger number of custom instructions that were identified for those three applications. For instance, even though the EDGE application also uses floating-point computation, only 3 custom instructions were identified for it, whereas 14 custom instructions were identified in total for the RT application. Therefore, the RT and VRC applications might not be good candidates for augmented issue-slot replication, despite the improvements in speedup shown in Fig. 2.22. The MPSO, in particular, is not a good candidate for augmented issue-slot replication at all, because the speedup remains the same for 2-AIS and 3-AIS configurations.

On the other hand, the CRC total and average energy requirements are very low in comparison to the energy requirements of the other applications. Thus, the CRC is a good candidate for instruction customization and further augmented issue-slot replication, especially if the improvement in speedup is considered. For the CRC, the tool automatically identified only one custom instruction that provides 14% speedup with low circuit area footprints. However, it has been shown in the previous Section that increasing the number of augmented issue-slots did not affect the speedup of the CRC application in particular. Therefore, the AES application may be a better candidate for augmented issue-slots replication: its circuit area and energy consumption footprints may be compensated by the higher speedup (from 16% to 35%) that can be achieved when using a pair of augmented issue-slots.

2.4.4 Fine-grain vs. coarse-grain analysis

Custom instructions are often organized into fine-grain and coarse-grain instructions [2]. The fine-grain instructions implement small groups of basic operations and are usually more general than the coarse-grain instructions, in the sense that they may be applied in an application domain. For instance, the multiply-accumulate is an example of fine-grain custom instruction that is often used in a broad range of multimedia applications. On the other hand, coarse-grain instructions implement large blocks of basic operations and are usually much less general than fine-grain instructions, i.e. they can hardly be used for more than one application.

This Section discusses the impact of the identified custom instructions granularity on the ASIP execution time, circuit area and energy consumption. Each maximum clique corresponds to a common sub-graph and the tool can automatically identify common operation patterns of different granularities between a set of basic blocks extracted from an application, or class of applications. The size of each enumerated clique determines the granularity of the corresponding generated custom instruction. For instance, a clique of size 2 generates custom instructions of 2 basic operations (*2-op*) and cliques of size 5 generates custom instructions of 5 basic operations (*5-op*).

The speedup in comparison to the default VLIW-ASIP instruction set implementation, i.e., without instruction set extensions, is presented in Fig. 4.13(a), for custom instructions of different granularities with 1,2 and 3 AIS, respectively. Further details can also be found in Table 2.5. The CI identification tool automatically sets the granularity of each custom instruction. The granularity represents the maximum size of a recurrent pattern identified by the tool for each application. For instance, for the JPEG application, custom instructions composed of 2,3 and 7 basic operations were identified, while for the ray-tracing application custom instructions composed of 2,3,4,5,6 and 8 basic operations were identified. Moreover, the speedup presented in a given higher granularity (*e.g. 5-op*) also includes the speedups of its respectively lower 2,3,4-op granularities, i.e. , the lower granularity custom instructions are also implemented.

It is possible to observe that most of the obtained speedup comes from the fine-grain custom instructions, in especial the *2-op* custom instructions, which are

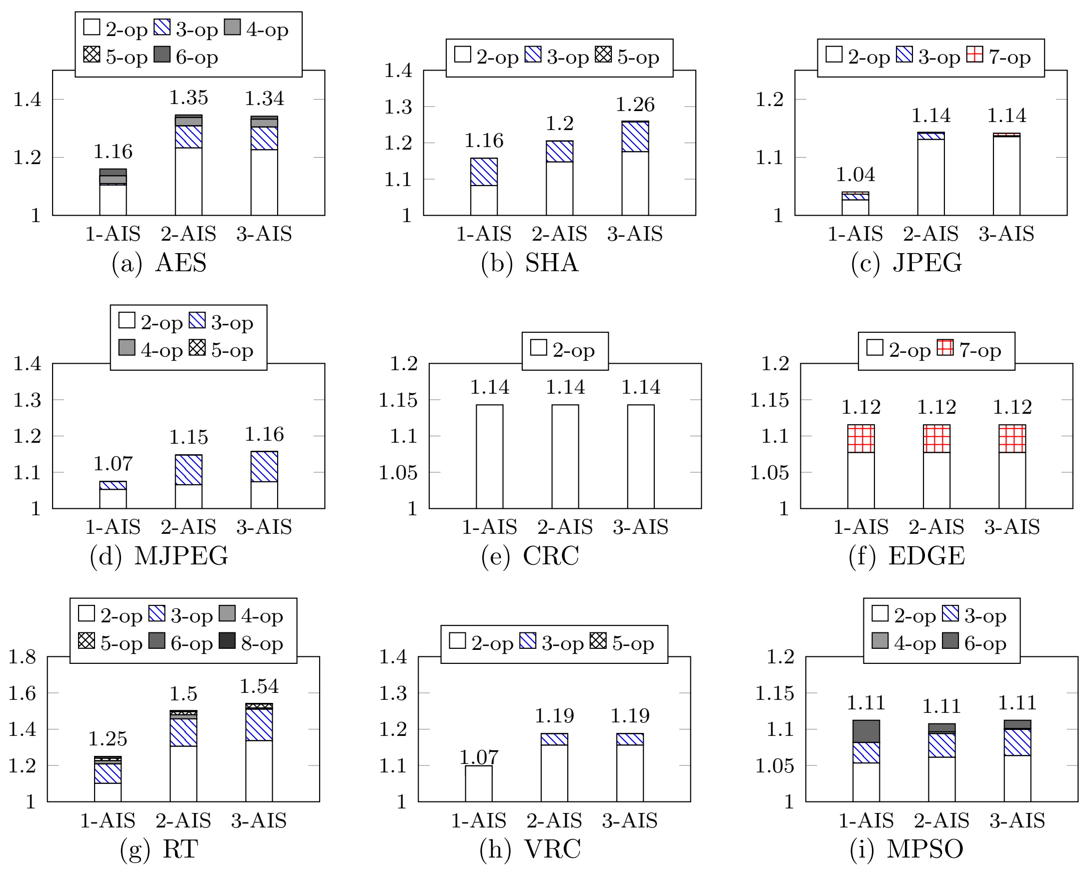


Figure 2.24: Speedup for each benchmark application using different automatically identified CI granularities with 1,2 and 3 augmented issue-slots (AIS), respectively, in comparison to using the standard instruction set.

composed of 2 basic operations only. For instance, when the custom instructions are enabled using 1-AIS, the *2-op* custom instruction identified for the CRC application provided alone a 14% speedup. Also, the RT application achieves a substantial speedup of 21% when *3-op* custom instructions are included together with *2-op* custom instructions, even though coarser-grain custom instructions have been identified. In fact, using coarser-grain custom instructions in the RT application increased the speedup to 25% only, which is close to the speedup achieved by the *3-op* custom instructions already. The same happens for several other applications, such as AES, JPEG and EDGE, where increasing the custom instructions granularity slightly affects (or even worsen, as in VRC) the speedup. The main reason for the fine-grain custom instruction speedup dominance over the others is the amount of times that the fine-grain custom instructions got mapped onto the application data-flow graph, as depicted in Fig. 2.25 and detailed in Table 2.5. It can be noted that fine-grain custom instructions are often mapped onto the application graph in comparison to coarser-grain custom instructions, especially for the AES, SHA, JPEG and MJPEG applications.

Furthermore, increasing the number of augmented issue-slots does not seem to

Table 2.5: Expanded results of speedup, circuit area and energy consumption for each custom instruction granularity.

		CI Speedup			CIs (1-AIS, TSMC 65nm)					
App.	#ops	1-AIS	2-AIS	3-AIS	#CIS	#Map	%Access	Area #gates	Avg.E. (pJ)	Total E. (nJ)
AES	2	1.10	1.23	1.23	3	261	11.35	3201	2.72	8895.52
	3	1.11	1.31	1.30	3	116	4.86	4817	4.06	5701.96
	4	1.14	1.34	1.33	1	9	0.0002	2080	5.38	0.32
	5	1.12	1.31	1.31	1	1	0.54	2488	6.75	1052.43
	6	1.16	1.35	1.34	1	8	0.54	3699	8.27	1289.43
SHA	2	1.08	1.15	1.18	3	45	3.56	3706	2.97	15809.11
	3	1.16	1.21	1.26	3	280	9.49	5315	4.35	61853.55
	5	1.16	1.20	1.26	1	1	0.03	3108	7.76	393.77
JPEG	2	1.03	1.13	1.14	7	429	14.65	30888	8.32	136995.81
	3	1.04	1.14	1.14	2	3	0.16	20354	18.53	3397.48
	7	1.04	1.14	1.14	2	6	0.49	31440	29.34	16052.11
MJPEG	2	1.05	1.07	1.07	2	530	2.62	2472	2.81	2451.64
	3	1.07	1.15	1.16	2	5	4.75	14918	13.93	22038.37
	4	1.07	1.15	1.16	1	3	0.0001	2145	5.49	0.12
CRC	1	1.14	1.14	1.14	2	3	18.18	984	2.7	143700.48
EDGE	2	1.08	1.08	1.08	2	7	1.58	7689	9.08	4446.62
	7	1.12	1.12	1.12	1	16	0.91	13862	33.89	9558.13
RT	2	1.10	1.31	1.34	3	35	1.02	39459	18.53	71619.89
	3	1.21	1.46	1.51	4	40	2.42	50064	30.68	282473.80
	4	1.23	1.48	1.52	3	35	0.78	50064	40.90	120745.90
	5	1.24	1.50	1.54	1	66	3.05	20205	48.83	565349.49
	6	1.25	1.50	1.54	2	12	0.82	48099	57.91	179791.72
	8	1.25	1.50	1.54	1	12	0.03	32721	79.51	9346.32
VRC	2	1.10	1.16	1.16	3	26	1.43	22116	16.23	801354.74
	3	1.07	1.19	1.19	3	11	6.74	35892	25.89	6005893.85
	5	1.07	1.19	1.19	1	2	0.01	20205	48.83	23438.40
MPSO	2	1.05	1.06	1.06	6	57	1.30	33784	13.16	114656.33
	3	1.08	1.09	1.10	3	45	1.73	24764	19.53	226872.92
	4	1.08	1.10	1.10	2	12	0.53	26826	29.42	105310.07
	6	1.11	1.11	1.11	2	33	0.64	26252	31.32	133450.26

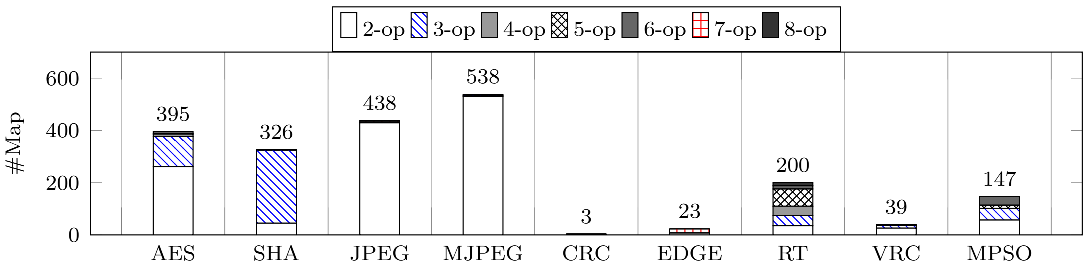
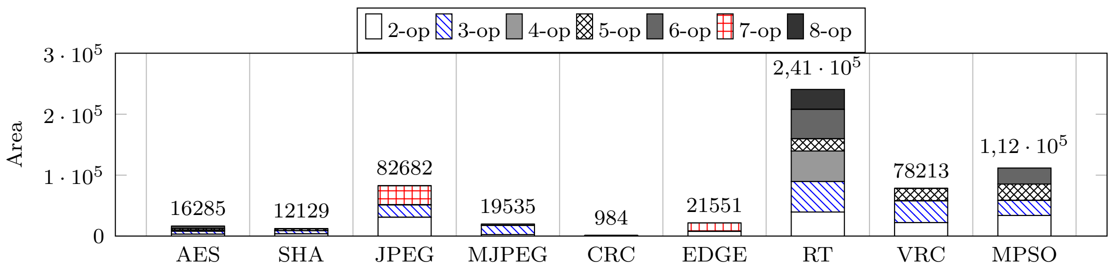


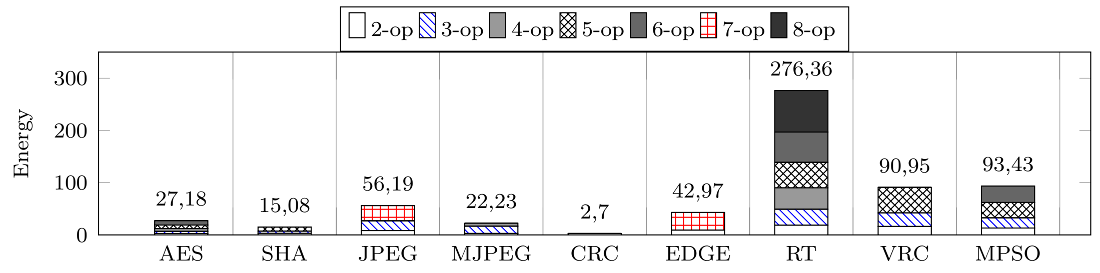
Figure 2.25: Mapping of each custom instruction onto the application data-flow graph.

significantly contribute to the (re)-distribution of the speedup among the different custom instructions, with the exception of a few applications, such as the MJPEG, RT and MPSO. Observe, for instance, that *3-op* custom instructions significantly helps to improve the performance of the MJPEG and RT applications when more augmented issue-slots are implemented. However, in the case of the MPSO application, using more augmented issue-slots did not affect the overall performance of the application, although it shifted the speedup gain from the *6-op* custom instructions back to *2-op* and *3-op* custom instructions.

Area and energy consumption estimative are given in Fig. 2.26, while specific details can also be found in Table 2.5 for a configuration of 1 AIS. It can be noted that the applications with the larger number of identified custom instructions are among the ones that require the most circuit area and, eventually, consume the most energy, such as the JPEG, RT, VRC and MPSO. Moreover, the three latter



(a) Circuit area estimation of each custom instruction.



(b) Average energy consumption estimation of the custom instructions.

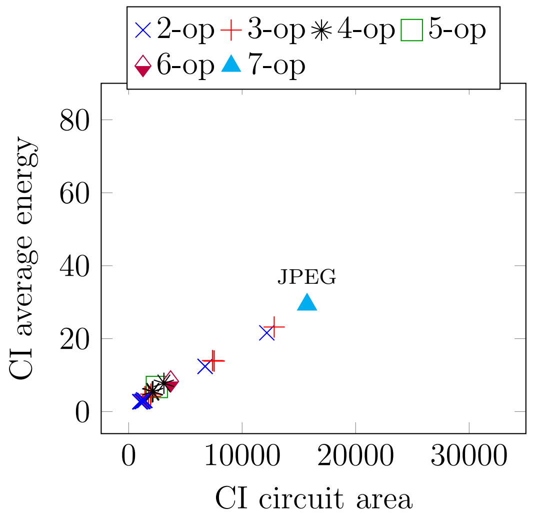
Figure 2.26: Custom instructions circuit area and energy estimation for different granularities and applications, using 1 augmented issue-slot and TSMC 65nm technology with 5ns timing constraint.

applications extensively use floating-point computation, which is known to require more area and consume more energy.

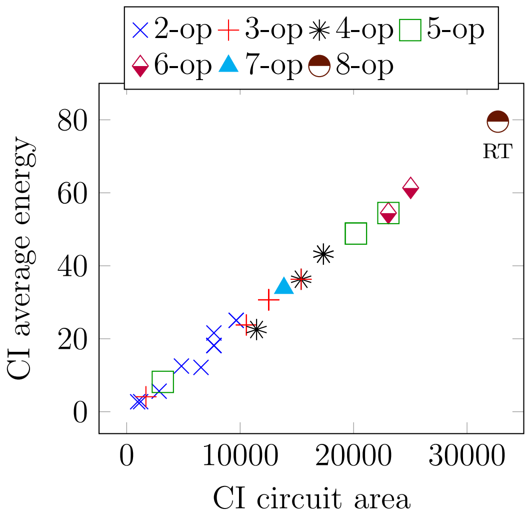
The distribution of the custom instructions granularities based on their circuit area and energy consumption is depicted in Fig. 2.27, organized into integer and floating-point custom instructions. Observe that the integer custom instructions are mostly concentrated in the lowest circuit area and energy consumption coordinates, regardless of its granularity, as shown in Fig. 2.27(a). For instance, *6-op*, *4-op* and *5-op* custom instructions are nearby several *2-op* custom instructions. Hence, they all share low area and energy consumption footprints, with exception of the *7-op* custom instruction, among a few others. However, the floating-point custom instructions, as shown in Fig. 2.27(b), are clearly less concentrated and can be distinguished from one another, with respect to their area and energy consumption. It can be noted that as the granularity increases, the higher becomes the area and the energy consumption of this type of custom instructions.

2.4.5 Comparison results

This Section evaluates the proposed custom instructions regarding execution time reduction and speedup in comparison to Wu's [32, 33] and Lu's [17] works, which are all based on multi-issue architectures. Table 2.6 summarizes the main differences between the works under comparison. In all the related works, different configurations of register files and/or issue-slots are provided. However, for the sake of



(a) 41 Integer custom instructions, with the mostly expensive belonging to the JPEG application.



(b) 29 Floating-point custom instructions, with the mostly expensive belonging to the RT application.

Figure 2.27: Distribution of all the 70 identified custom instructions over their circuit area and energy consumption.

fairness, we only present the results from the configurations which are similar or equivalent to ours. Some results, such as the number of custom instructions, execution time reduction and speedup, are averaged due to the different set of benchmark applications used in each work.

Table 2.6: Summary of the main differences between the proposed and related works.

	Wu 2008	Lu 2008	Wu 2014	Proposed
Architecture	VLIW	TTA	VLIW	VLIW
Issue-width	3	4	3	3
Register file R/W ports	8/4	6/2	8/4	6/1
# CIs	32	N.A.	N.A.	10
Avg. Speedup	N.A.	1.73	1.58	1.23
Execution time reduction	14.35%	N.A.	N.A.	18.025%
Technology	0.13um	0.18um	0.13um	65nm
Methodology	Ant Colony Optimization	Minimal Length Covering	EI Exploration	Maximal Clique Enumeration
Benchmark	adpcm, basicmath, epic, fft, inverse-fft, g.721, jpeg, mpeg2, sha, susan	sha, blowfish, cjpeg, djpeg, gsmencode, gsmdecode, dijkstra, mpeg2enc, mpeg2dec, rijndael	adpcm, bitcount, blowfish, crc32, dijkstra, rijndael, stringsearch	aes, sha, jpeg, crc32, mjpeg, edge, rt, vrc

First of all, it is important to highlight that, even though all the related works compared here exploit a multi-issue architecture to evaluate their proposals, none of them provided details about the used architectures, as in Section 2.4.1 and Fig. 1.2. For instance, in Wu's most recently published work [33], the results are validated using an in-house-developed cycle-accurate VLIW simulator, with no further details regarding the underlying architecture, while Lu's work [17] limits to pointing to the Transport Triggered Architecture (TTA) compiler toolchain that was used, abstracting the details about the underlying architecture. Furthermore, aside from circuit area results, none of the compared works provide energy consumption results of the identified custom instructions, nor their usage frequency.

Wu's first work [32] proposed an algorithm for instruction set extension based on list scheduling and Ant Colony Optimization (ACO), achieving an average execution time reduction of 14.35%. In this case, the instruction set is extended with 32 new instructions, exploiting an issue-width of 3 and a register file of 8 read ports and 4 write ports. The proposed method achieves 18.025% reduction, using less custom instructions and less read/write ports, listed in Table 2.6. Later, Wu proposed an EI (Extended Instruction) exploration algorithm [33]. Several configurations are provided yielding different speedup results. However, for comparison reasons, we selected the configuration that is most compatible to the proposed here. Thus, using an issue-width of 3 and a register file of 8 read ports and 4 write ports, Wu achieves an average speedup of 1.58, when the extended instructions are enabled, against a speedup of 1.23 as achieved by the proposed method. Despite Wu's higher speedup, it is not possible to establish the superiority of his approach with regards to the one proposed here due to the absence of details regarding the number of custom instructions that were used. Also, considering the register file configuration, it is possible to infer that the custom instructions in Wu's proposal can have multiple outputs, probably because the selected subgraphs therein are disconnected. This may increase the custom instruction parallelism.

Finally, Lu's work [17] proposed an algorithm focused on instruction selection, which is a graph covering problem and, thus, different from the custom instruction identification method proposed in this work. Nonetheless, sets of custom instructions were identified using traditional convex cuts [27]. The average speedup is the highest amongst all compared works. However, from Table 2.6, it is possible to observe that very few details are given about the exploited TTA architecture configuration, which makes the comparison of the results more difficult. The only available details are the issue-width and the number of register file I/O ports. The issue-width is higher than that used in the proposed configuration and the number of output ports also suggests that the custom instructions in Lu's proposal can have multiple outputs, as in Wu's work. Altogether, these two characteristics might have caused the reported higher speedup. Also, the TTA architecture is known as a special case of VLIW architecture, with potentially higher Instruction Level Parallelism. This may also have influenced the speedup.

Chapter 3

Hardware Sharing

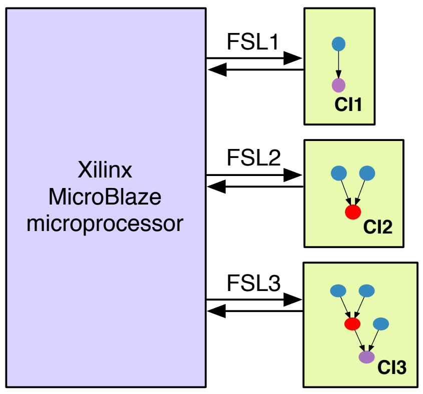
Hardware sharing, a.k.a hardware reuse or resource sharing, is a well-known circuit and system optimization approach traditionally employed for saving circuit area [4, 48, 49]. It is an important and widely used area-reduction technique that can be employed in different stages of the digital system design processes, such as in the application-specific instruction set extension [19] or high-level synthesis resource binding [50, 51]. Also, many commercially available synthesis tools can perform some degree of resource sharing in their RTL circuit area optimizations [49, 52].

In high-level synthesis, the process of resource binding consists of assigning operators and variables from the high-level specification to hardware function units and registers. In this case, resource sharing occurs when multiple operators are assigned to the same function unit and scheduled at different cycles [51]. Thus, high-level synthesis often performs resource sharing within the same datapath. Also, there is usually a substantial hardware sharing potential in the input description for the RTL compiler, *e.g.*, simple RTL hardware description changes can greatly affect the way a given design is synthesized. For instance, the mutually exclusive execution branches can be explicitly identified through transformations of the original conditional description, improving the resource sharing potential [53].

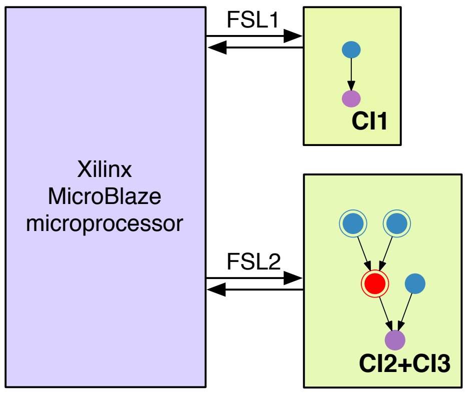
Moreover, in parallel to the creation of new important opportunities, the recent progress in the semiconductor technology, and particularly the introduction of the nano-dimension CMOS technologies, has created many difficult to solve problems. System and circuit complexity, energy and power issues, the interconnect scalability problems and on-chip communication issues are some of the most important. Moreover, the introduction of the nano-CMOS technologies changed the importance relationships among various design aspects. For instance, the static power negligible in the past is now comparable to its dynamic counterpart, and the interconnects, instead of active elements, tend to have a dominating influence on the major SoC characteristics (area, throughput, etc.). On the other hand, many modern mobile/autonomous applications in several fields like communications, multimedia, security, military and medical instruments impose extremely high throughput and/or energy related requirements.

Hardware reuse can be exploited in different stages of the system development [50, 51], in particular during the instruction set extension [20, 54, 55]. In this stage, hardware sharing becomes an ever more important optimization due to the often high circuit area requirements imposed by the new custom instructions, especially if these require floating-point function units, as shown in Section 2.4.3. Observe that the introduction of custom instructions into the ASIP datapath can actually be interpreted as a replication of function units, in order to speedup the execution of an specific application or class of applications. Thus, the key aspects of resource sharing is to increase the ASIP/hardware accelerator performance through hardware implementation of custom instructions/functions and at the same time avoid replicating logic, *e.g.*, adders, subtractors, multipliers, etc. In both cases, resource sharing may increase the circuit latency, especially because of the insertion of multiplexers that are used to share common paths of the datapath.

In the context of datapath synthesis of custom instructions, hardware sharing optimizations typically take place among operations that are mutually exclusive, *i.e.*, operations that are never executed at the same time. Otherwise, such area saving optimization can reduce the overall system's performance, because the operations would have to compete for the shared resource, *e.g.*, function unit, and would no longer be executed in parallel. This is often the case for instruction set extensions that are implemented as co-processors, as depicted in Fig. 3.1. It can be noted that each custom instruction – CI1, CI2 and CI3 – can be executed in parallel in Fig. 3.1(a), whereas the two merged custom instructions, CI2 and CI3, in Fig. 3.1(b) cannot be executed in parallel anymore.



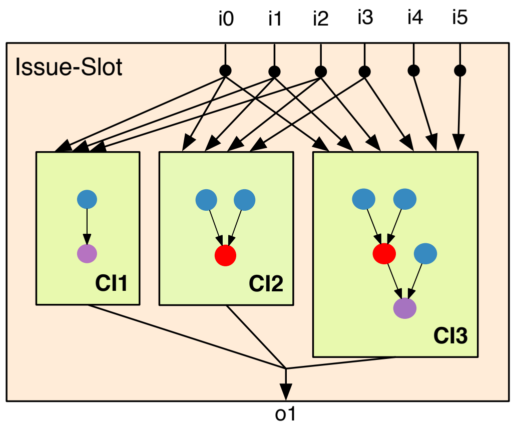
(a) MicroBlaze microprocessor connected to dedicated co-processors.



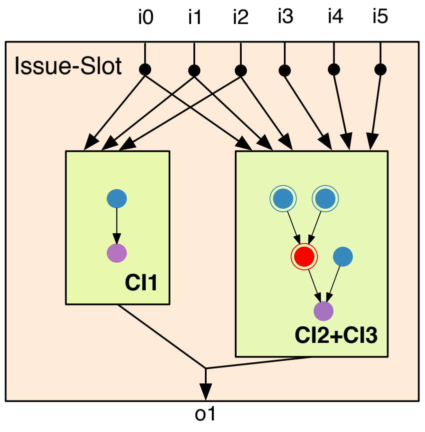
(b) MicroBlaze microprocessor connected to a shared co-processor and one dedicated co-processor.

Figure 3.1: Customization example of the Xilinx MicroBlaze microprocessor instruction set with hardware sharing optimization.

On the other hand, the problem of mutual exclusiveness is not as severe as in the case of VLIW-based ASIPs, because the function units that reside in the same issue-slot are inherently mutual exclusive: an issue-slot can only operate on one function unit at a time, leaving the others unused. Thus, there is a high potential for hardware sharing inside an issue-slot, as shown in Fig. 3.2.



(a) VLIW-ASIP issue-slot with dedicated custom function units.



(b) VLIW-ASIP issue-slot with one dedicated and one shared custom function units.

Figure 3.2: Customization example of the VLIW-ASIP instructions set in one issue-slot.

Unfortunately, contemporary circuit and system design methods and tools remain behind the actual needs of modern applications and technologies. In particular, they do not well account for the changed importance relationships of different design aspects, and for the now necessary multi-objective decision modeling and careful tradeoff exploitation among various design objectives. This particularly relates to the resource sharing aspects that can be exploited to limit the system and circuit area, but equally well, to limit the system and circuit energy consumption, and in some specific cases, even to increase the speed. This way, hardware reuse is one of the major aspects of the multi-objective system and circuit optimization and tradeoff exploitation.

This Chapter discusses the problem of hardware reuse in synthesis of ASIPs and hardware accelerators implemented in modern nano-CMOS technologies. It analyses the problem and presents important results of high practical relevance from extensive experimental research regarding this issue at the Register Transfer Level (RTL). It demonstrates that the state-of-the-art automatic synthesis tools do not address this problem in a sufficient way, and shows how important an effective hardware reuse is for an adequate synthesis of ASIPs and hardware accelerators. Furthermore, datapath merging techniques are compared and a clique-based approach is proposed as candidate for reducing the circuit area and, possibly, reducing the energy consumption of the identified custom instructions, similar to the method described in Chapter 2.

3.1 RTL compiler hardware sharing

Resource sharing at the register transfer level works by sharing one or more function units (FUs) to implement several operations described in HDL, producing less hardware components in the final *netlist* representation. Therefore, resource sharing may also substantially reduce the energy consumption.

Despite various hardware sharing optimizations performed at the earlier micro-architecture synthesis stages, there is usually a substantial further hardware sharing potential in the input description for the RTL compiler. Moreover, simple RTL hardware description changes can greatly affect the way a given design is synthesized. For instance, changing conditional statements, such as *IF* or *CASE* statements, may produce a different circuit [4]. The mutually exclusive forms are one of the basis for successful resource sharing. In [53], independent of which conditional construction is used, the authors proposed to explicitly identify the mutually exclusive execution branches through transformations of the original conditional description. In the end, a larger set of mutual exclusive branches is detected, allowing a better reuse of resources. Consider first a simple example presented in Listing 3.1.

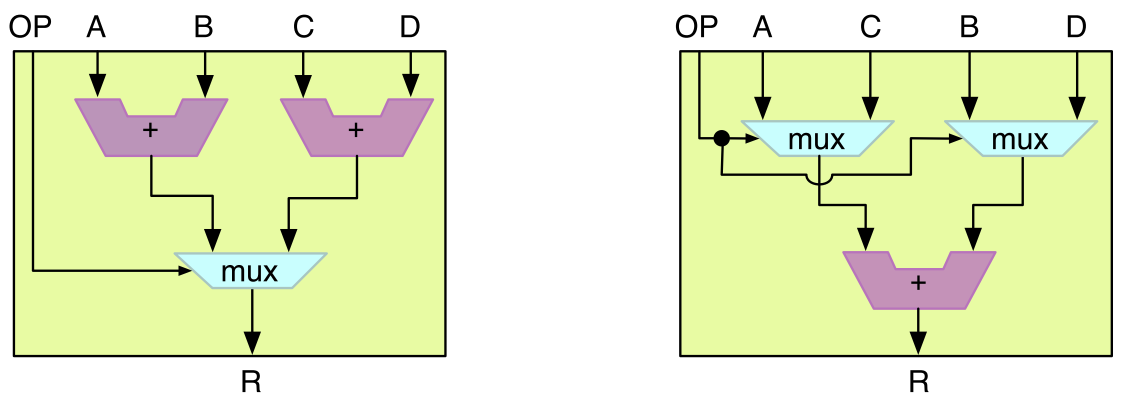
Listing 3.1: VHDL architecture example for two add operations.

```vhdl
architecture behavioral of FU is begin
process(A,B,C,D,OP)
begin
  if(OP = '0')then
    R <= A + B;
  else
    R <= C + D;
  end if;
end process;
end behavioral;
```

Without resource sharing optimizations, the synthesis tool would create separate circuits, *i.e.*, adders, one for each operation, as in Fig.3.3(a). However, it is possible to allow resource sharing transformations, provided that both operations are never used at the same time, as depicted in Fig.3.3(b).

3.1.1 Automatic resource sharing

Most commercially available synthesis tools are able to automatically detect and exploit some limited resource sharing possibilities within a digital design [49], as for instance shown in the example of Fig.3.3, based on Listing 3.1. Usually, automatic



(a) No shared resources.

(b) Shared resources.

Figure 3.3: Resource sharing example. The two addition operations are never going to be executed at the same time, because the operation selector *OP* ensures that they are mutually exclusive.

resource sharing is enabled whenever the synthesis tool is set to a high area reduction effort optimization mode and operations are never executed at the same time [53]. Furthermore, in Fig.3.3 it is possible to observe that applying resource sharing caused an exchange of a relatively complex function unit (adder) for a simpler extra multiplexer. This is usually the case, because sharing of one function unit among many operations implies the usage of multiplexers to select which input data to process, which also may introduce the cost of a possible additional latency [4]. If timing constraints cannot be met, the synthesis tool may disable resource sharing in advance.

3.1.2 Manual resource sharing

The synthesis tools are often not capable of automatically exploit the sharing opportunities, mainly because of the following three reasons:

1. The hardware is not specified in an appropriate style, so that the tool is unable to identify sharing opportunities;

2. The tool misses more global and extensive application analysis, and only exploits some local optimizations for resource sharing;

3. Resource sharing is not possible due to timing constraints issues.

The first issue can often be solved by rewriting the hardware description, so that resource sharing opportunities are better visible to a particular tool. For instance, instead of performing an operation inside of each conditional statement, the input data should be first selected. Only after the evaluation of the boolean expressions and input data selection, the operation is performed, as shown in Listing 3.2.

Listing 3.2: Rewriting Listing 3.1 architecture to ensure resource sharing.

```
architecture behavioral of FU is begin
process(A,B,C,D,OP)
var X,Y : std_logic_vector(N-1 downto 0);
begin
  if(OP = '0')then
    X <= A;
    Y <= B;
  else
    X <= C;
    Y <= D;
  end if;
  R <= X + Y;
end process;
end behavioral;
```

The second issue can only sometimes be solved by specification rewriting, but often requires a different synthesis method and tool extensions. The third issue cannot easily be solved by only rewriting the specification. It requires a careful analysis of the digital design in order to perform complex algorithm optimizations that could reduce the digital system latency, such as the replacement of Ripple-Carry adders by faster Carry-Save adders [4, 56, 57].

For those reasons, commercially available synthesis tools often provide some general guidelines that make possible to achieve resource sharing optimizations. The VHDL hardware description in Listing 3.3 is used as an example. In essence, a resource sharing for two or more operations is only possible if the operations are never going to be executed at the same time, as presented in Listing 3.3. Otherwise, serialization of operations must take place. In this example, operations (A+B) and (C+D) can share a single adder within the first process, but cannot share the same adder across with the second process, because the second process can be executed concurrently with first one.

Also, note that operations (X+Y) and (Z-W) can be shared within the second process, because they are similar: a subtraction can be implemented as an addition with the second operand inverted and with incoming carry equal to one in the lowest position [56]. Thus, the synthesis software should create in this situation a single adder function unit and inverter for the second operand to enable subtraction. This kind of resource sharing is also referred as *Functionality Sharing* and is more difficult to be achieved automatically, because it requires a deeper knowledge of the relationships between the operations (or functionalities) that can be merged into a single hardware. Further examples can be found in [4, 48].

Nevertheless, in general, placing operations considered for possible sharing more explicitly at mutually exclusive branches in the RTL hardware specification might facilitate the identification of the resource sharing potential. For this reason, the use of conditional *CASE* statements should be preferred, because the case branches

Listing 3.3: Architecture description with mutually exclusive statements and separate processes.

```
architecture behavioral of FU is begin
--First process
P1: process(A,B,C,D,OP1)
begin
   if(OP1 = '0')then
     R1 <= A + B;
   else
     R1 <= C + D;
   end if;
end process;
--Second process
P2: process(X,Y,Z,W,OP2)
begin
   if(OP2 = '0')then
     R2 <= X + Y;
   else
     R2 <= Z - W;
   end if;
end process;
end behavioral;
```

have to be mutually exclusive. A similar principle applies for operations described in separate processes: inside the process, statements are executed sequentially, but two or more processes are executed in parallel.

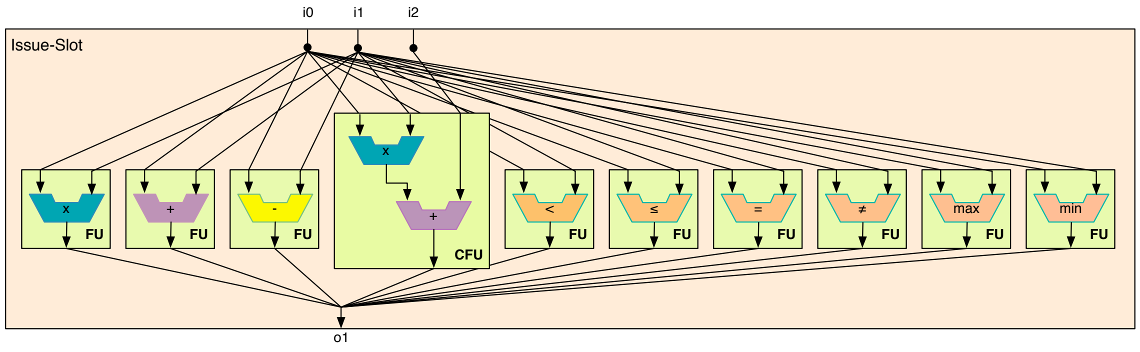
3.1.3 Experimental results on automatic resource sharing

An example of RTL compiler automatic resource sharing is given in this Section for several function units of one issue-slot of the VLIW-ASIP, as depicted in Fig. 3.4(a), using Cadence logic synthesis tools. Each function unit execute an operation presented in Table 3.1. The objective of this experiment is to evaluate how far the synthesis tool can handle hardware sharing optimizations when the operations are implemented as one function unit, as shown in Fig. 3.4(b).

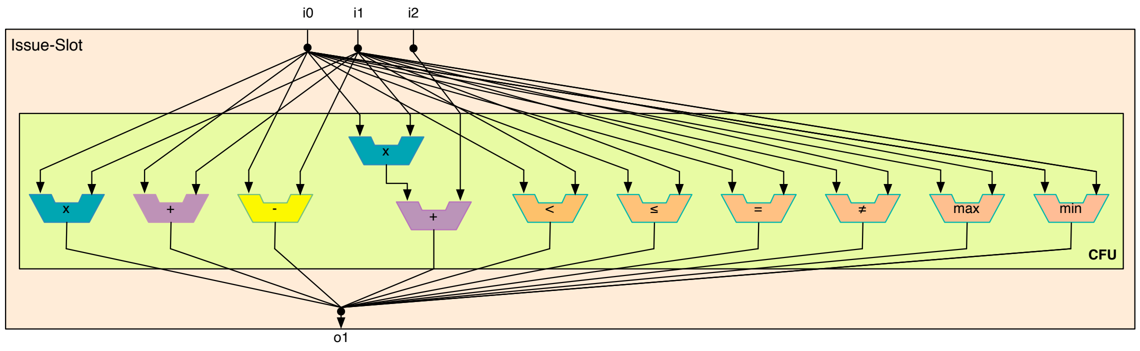
Table 3.1: Table of operations of each function unit of the VLIW-ASIP.

OPERATION	OPERANDS
Addition	$o_1 <= i_0 + i_1$
Subtraction	$o_1 <= i_0 - i_1$
Multiplication	$o_1 <= i_0 \times i_1$
Multiply–Accumulate	$o_1 <= i_0 \times i_1 + i_2$
Lower equal	$o_1 <= (i_0 \leq i_1)$
Lower than	$o_1 <= (i_0 < i_1)$
Equal to	$o_1 <= (i_0 = i_1)$
Not equal to	$o_1 <= (i_0 \neq i_1)$
Maximum	$o_1 <= max(i_0, i_1)$
Minimum	$o_1 <= min(i_0, i_1)$

The synthesis results are summarized in Table 3.2, for a process of 90nm and a timing constraint of 10.0 ns. The medium area reduction effort configuration



(a) Each operator implemented into one separate function unit.



(b) All the operators implemented together into one function unit.

Figure 3.4: Operators implemented as separate function units or as a single function unit of an issue-slot of the VLIW-ASIP.

disables the hardware sharing optimizations. Therefore, a high area reduction effort has been enabled in order to activate the synthesis tool automatic resource sharing capabilities. Thus, instead of producing one arithmetic and one logic operator for each operation of Table 3.1, the resource sharing compiler optimization produced only three arithmetic and logic operators in total. For instance, two adders and one subtractor were merged into a single arithmetic unit, while all the logic operations were merged into a single logic unit, with the exception for the *max* and *min* logic units. The remaining multipliers were not merged too. Thus, around 6% of area reduction has been achieved for this example of resource sharing. The optimized function unit is depicted in Fig. 3.5.

Table 3.2: Issue-slot function unit synthesis results.

Effort	Cell Area	Net Area	Total Area	Delay (ps)
Medium	19764	14493	34257	6462.0
High	18550	13388	31938	6367.0
Reduction Rate	6.14%	7.62%	6.77%	1.47%

However, resource sharing optimizations may become more difficult for even

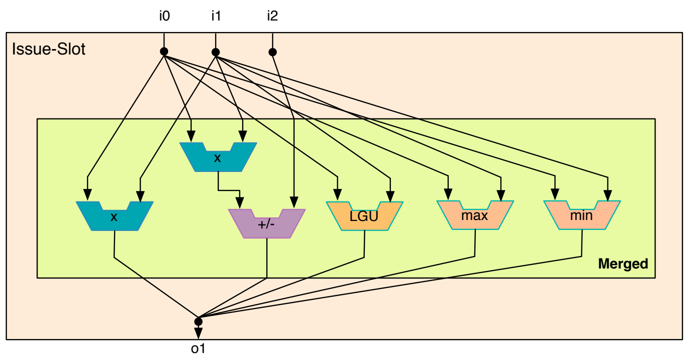
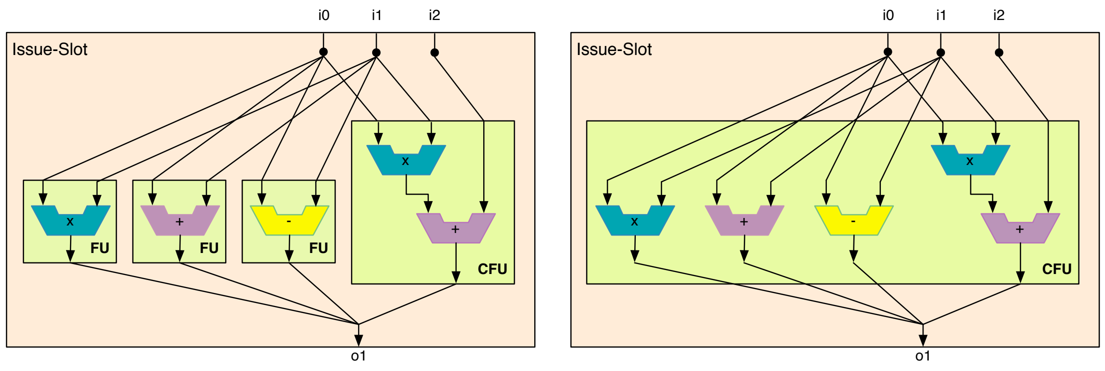


Figure 3.5: Resulting merged operators.

lower nano-CMOS dimensions, lower timing constraints and using different RTL specifications styles. Therefore, further results are presented in the following Section for the synthesis of a different function unit, using a technology of 40nm and a timing constraint of 5ns.

3.1.4 Experimental results on manual resource sharing

In this Section, some selected results are discussed for the manual hardware sharing at the RTL of several function units of one issue-slot of the VLIW-ASIP, as shown in Fig. 3.6. The experimental results are based on the synthesis of a function unit that performs four similar operations, which are described in Table 3.3. Four hardware architecture description styles of the same function unit entity are presented, so that resource sharing potential identification and exploitation can be analyzed for each case. The first description style is a mixed structural and behavioral, while the others are purely behavioral. Also, the selected operations are similar to each other, so that they may be merged into single operators, such as an adder/subtractor.



(a) Each operator implemented as one separate function unit.

(b) All the operators implemented together into one function unit to enable hardware sharing.

Figure 3.6: Function units of an issue-slot of the VLIW-ASIP.

The operations cannot be executed at the same time. Its computation result is

Table 3.3: Table of operations of each function unit of the VLIW-ASIP.

OPERATION	OPERANDS
Addition	$o_1 <= i_0 + i_1$
Subtraction	$o_1 <= i_0 - i_1$
Multiplication	$o_1 <= i_0 \times i_1$
Multiply–Accumulate	$o_1 <= i_0 \times i_1 + i_2$

produced based on the selected operation and input data. Since they are mutually exclusive, they can be shared. In the following sections, the synthesis results of the function unit shown in Fig. 3.6(b) are discussed for each hardware description style and area reduction effort (medium or high), with the high area reduction effort enabling automatic resource sharing. First, each architectural description style of the function unit is presented and discussed together with the corresponding synthesis results. All the presented results were collected using Cadence logic synthesis compiler for a TSMC 40nm technology and a timing constraint of 5ns.

3.1.4.1 Structural RTL description

The first architecture description style is a structural description of the function unit, and it corresponds to a particular optimal circuit that is expected to be created. However, each sub-component of this structural description was described in a behavioral style, so that the synthesis software is in charge of creating or selecting an appropriate final hardware.

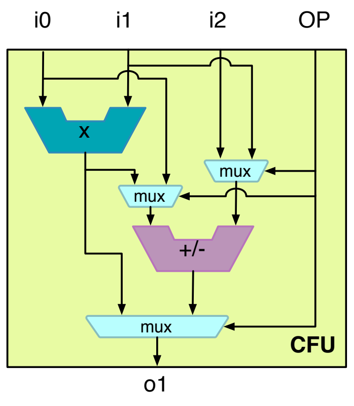


Figure 3.7: Structural design.

In this structural description, the synthesizer was able to precisely identify each component as they were described, as depicted in Fig. 3.7. Also, a high area reduction effort exploited some resource sharing by merging the adder and the subtractor into a single hardware. The results are presented in Table 3.4, for both area reduction efforts. Observe that the overall area is greatly reduced thanks to resource sharing. However, the resource sharing optimizations can increase the worst path delay, because of extra multiplexers that are included to select between different input data, as depicted in Fig. 3.7.

Table 3.4: Structural description synthesis results.

Effort	Cell Area	Net Area	Total Area	Delay (ps)
Medium	3717	8944	12661	4025.7
High	3151	8198	11349	4126.0
Reduction Rate	15.23%	8.34%	10.36%	-2.5%

3.1.4.2 Conditional case statements

In the second architecture description style, as well as in the following architecture descriptions presented in Sections 3.1.4.3 and 3.1.4.4, the hardware is purely described in behavioral style. In the architecture speculation style of Section 3.1.4.2, conditional case statements are used to select between one of the four operations, according to the architecture description in Listing 3.4.

Listing 3.4: Architecture description using case statements.

```
architecture example of FU is begin
process(A,B,C,OP)
begin
  case OP is
    when MUL =>
      R <= A * B;
    when MAC =>
      R <= A * B + C;
    when SUB =>
      R <= A - B;
    when others =>
      R <= A + C;
  end case;
end process;
end example;
```

As expected, with the high area effort, the resource sharing was also performed for both adders and the subtractor operation, merging them into a single hardware unit. Also, only one multiplier was created, instead of two, as depicted in Fig. 3.8, and the multiplexers were replaced by an selection logic in the logic synthesis stage. On the other hand, disabling resource sharing possibility through not selecting the high area effort produced more hardware blocks, as shown in Table 3.5. Additionally, carry-save adders were included into the design, which greatly contributed to increase the Net Area. Also, timing constraints were not met for a medium area reduction effort, possibly because of the longer paths due to the higher Net and Total Area. Although the overall area for this design style with a high area reduction effort has been reduced, it is still larger than the area of the structural design of Section 3.1.4.1 under the same reduction effort. This shows that the synthesis tools did not realize an adequate resource sharing as expected from this design style. Regarding the worst path delay, a small improvement can be observed, as a result of using optimized selection logic instead of multiplexers in the inputs, as in Fig. 3.8.

Table 3.5: Case statements description synthesis results.

Effort	Cell Area	Net Area	Total Area	Delay (ps)
Medium	3850	33120	36970	4161.7
High	3254	8368	11622	4067.3
Reduction Rate	15.48%	74.73%	68.56%	2.26%

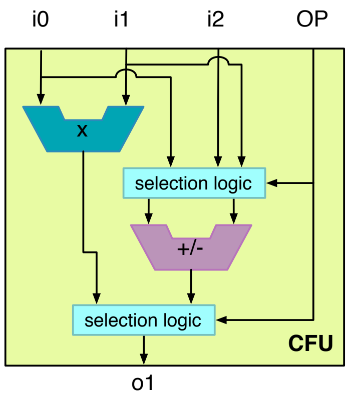


Figure 3.8: Case statement design style.

3.1.4.3 Conditional if statements

Since the usage of conditional *case* statements produced worse than expected results, the architecture description style was changed from *case* to *if* conditional statements. Also, following the example presented in Listing 3.1 to enhance the possibility of resource sharing, the inputs are first selected, so that the operation is executed afterwards, as shown in Listing 3.5. Such behavioral description is an attempt to achieve a more substantial resource sharing, aiming at producing a design as close as possible to the structural one depicted in Fig. 3.7.

From the results presented in Table 3.6, it is possible to observe that even though a high area reduction effort produced a slightly lower Cell Area compared to the medium effort, the Net Area increased substantially, in contrast to the results using the *case statements* in Section 3.1.4.2. Thus, a high area reduction effort created a worse design regarding the Net Area and could not find resource sharing between any of the operations. One possible explanation is that the design is already described in a style aware of resource sharing. Therefore, applying a high reduction effort

Listing 3.5: Architecture description using if statements.

```
architecture example of FU is begin
process (A,B,C,OP)
  variable var_mlt:std_logic_vector(2*N-1 downto 0);
  variable v_A,v_B:std_logic_vector(N-1 downto 0);
  variable output :std_logic_vector(N-1 downto 0);
begin
  var_mlt := A * B;
  if(OP = MAC) then
    v_A := var_mlt(N-1 downto 0);
    v_B := C;
  else
    v_A := A;
    v_B := B;
  end if;
  if(OP  = MUL) then
    output := var_mlt(N-1 downto 0);
  elsif(OP = SUB) then
    output := v_A - v_B;
  else
    output := v_A + v_B;
  end if;
end process;
end example;
```

introduced further optimizations that worsen the design. For instance, carry-save adders were included by the synthesis tool, which could contribute to increase the Net Area, as depicted in Fig. 3.9. As it can be observed, a separate carry-save adder was created for almost each operation and the multiplexers were also exchanged for optimized logic. The interconnections between them are simplified in Fig. 3.9. On the other hand, a medium area reduction effort yielded a better Net Area result, because no carry-save adders were included with this style, and no function merging of the adder and the subtractor was performed at all.

Table 3.6: If statements description synthesis results.

Effort	Cell Area	Net Area	Total Area	Delay (ps)
Medium	3740	9739	13479	4062.5
High	3689	10497	14186	3939.4
Reduction Rate	1.36%	-7.78%	-5.25%	3.03%

As it can be observed from Table 3.6, the worst path delay in this design style is better than the worst path delay in the previous design examples. As expected, the carry-save adders were able to increase the design's speed, while the Net Area also increased.

3.1.4.4 Cascaded conditional if statements

Finally, the architecture description from Listing 3.5 was slightly modified to the one presented in Listing 3.6, with the only difference being the cascaded (or nested)

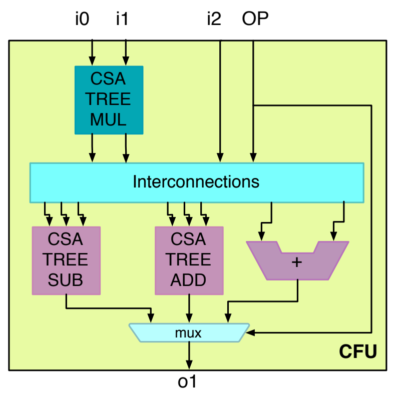


Figure 3.9: If statement design style.

if statement for selecting between the addition and subtraction operations. The idea is to keep these two operations close to each other in one internal *if-else* block of the nested if structure, so the synthesizer can more easily identify the possible resource sharing between them, because in the previous example it clearly could not identify the resource sharing potential.

Listing 3.6: Architecture description using cascaded if statements.

```
architecture example of FU is begin
process (A,B,C,OP)
  variable var_mlt:std_logic_vector(2*N-1 downto 0);
  variable v_A,v_B:std_logic_vector(N-1 downto 0);
  variable output :std_logic_vector(N-1 downto 0);
begin
  var_mlt := A * B;
  if(OP = MAC) then
    v_A := var_mlt(N-1 downto 0);
    v_B := C;
  else
    v_A := A;
    v_B := B;
  end if;
  if(OP  = MUL) then
    output := var_mlt(N-1 downto 0);
  else
    if(OP = SUB) then
      output := v_A - v_B;
    else
      output := v_A + v_B;
    end if;
  end if;
end process;
end example;
```

Once again the results were unsatisfactory, especially regarded the Net Area, as shown in Table 3.7. Carry-save adders were also included at the cost of an additional Net Area requirement. In contrast to the *if statement* design example, this design style actually helped the synthesis tool to identify the resource sharing between the adder and the subtractor operators, again merging them into a single hardware. The worst path delay is the best among all the others, even though the Net Area has increased substantially and no carry-save adders were introduced. The circuit schematic is very similar to the schematic of the *case* design in Fig. 3.8, with a few additional optimizations in the multiplexers.

Table 3.7: Cascaded if statements description synthesis results.

Effort	Cell Area	Net Area	Total Area	Delay (ps)
Medium	3525	8509	12034	4148.8
High	3576	16071	19647	3883.4
Rate	-1.45%	-88.9%	-63.26%	6.40%

3.1.4.5 Floating-point adder

This Section presents the synthesis results for a floating-point adder and subtractor [58]. This floating-point unit is divided in 3 basic stages: pre-normalization, operation core and post-normalization. The objective is to evaluate whether the synthesis tool is capable of discovering resource sharing in a more complex design, such as in a floating-point adder/subtractor. As expected, the synthesis tool was able to discover a resource sharing opportunity inside the arithmetic core, in particular at the addition/subtraction description in Listing 3.7, producing only one adder/subtractor unit. Hence, one of the steps of a floating-point addition consists of a simple integer addition of the fraction part from the input operands. The results are presented in Table 3.8. It is possible to observe that the overall area has been reduced and the worst path delay has been improved, even with resource sharing enabled.

Listing 3.7: Floating-point fraction integer addition/subtraction.

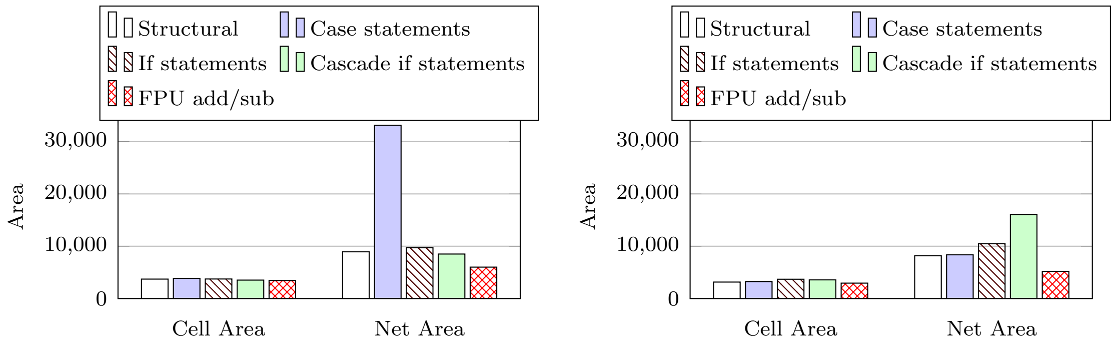
```
process(s_fracta_i, s_fractb_i, s_addop,
        fracta_lt_fractb)
begin
  if s_addop='0' then
    s_fract_o <= s_fracta_i + s_fractb_i;
  else
    if fracta_lt_fractb = '1' then
      s_fract_o <= s_fracta_i - s_fractb_i;
    else
      s_fract_o <= s_fractb_i - s_fracta_i;
    end if;
  end if;
end process;
```

3.1.5 Results overview

The experimental results are summarized in Table 3.9 and in Fig. 3.10, for a process of 40nm and a clock period of 5.0 ns. First of all, it is clear that the structural architecture description, with the implementation structure imposed by a human designer to a high degree, is the one that produced the best results with high area

Table 3.8: Floating-point Adder.

Effort	Cell Area	Net Area	Total Area	Delay (ps)
Medium	3456	6005	9461	4322.4
High	2946	5184	8130	4218.7
Reduction Rate	14.75%	13.67%	14.07%	2.4%



(a) Medium area reduction effort. (b) High area reduction effort.

Figure 3.10: Area synthesis results for different specification styles.

reduction effort, as shown in Fig. 3.10(b). To get the high-quality results, a structural description style is usually preferred over a behavioral one, because a human designer or a higher-level architecture synthesis tool can more directly and precisely specify the required design features and such description can be easier translated by the synthesis tool to a corresponding high-quality netlist. Also, enabling the resource sharing via a high area reduction effort clearly proved to be useful for saving resources, and resulted in substantial area improvements, with a Cell reduction area ratio of 15.23% and Net reduction area ratio of 8.34%, for such a structural design. The hardware sharing also reduced the overall area in different design styles. Furthermore, if only the medium area reduction effort is considered, then the best area result is achieved by the last set of experiments: the *cascaded if statements*. Such description style was able to improve the area cost, because the architecture was described in the style easier for resource sharing identification. Further optimizations performed with a high area reduction effort mainly included additional circuitry to the design and increased the Net Area, but reduced the worst path delay, as shown in Fig. 3.11.

The resource sharing result very much depends on the hardware description style, as also observed in Table 3.9. For some design styles, a substantial increase in Net Area can be observed, especially due to further timing optimizations. The inclusion of carry-saver adders in several cases greatly increased the Net Area requirements, in an effort to overcome the extra delays that might arise from hardware sharing.

Table 3.9: Synthesis results, for different design styles and area reduction efforts (TSMC 40nm process and 5ns timing constaint).

Design style	Structural		Case statements		If statements		Cascade if statements		FP add/sub	
	Area	Delay	Area	Delay	Area	Delay	Area	Delay	Area	Delay
Medium	12661	4025.7	36970	4161.7	13479	4062.5	12034	4148.8	9461	4322.4
High	11349	4126.0	11622	4067.3	14186	3939.4	19647	3883.4	8130	4218.7
Rate*	-10.36%	+2.5%	-68.56%	-2.26%	+5.25%	-3.03%	+63.26%	-6.40%	-14.07%	-2.4%

*A plus signal before **rate** data actually indicates an increase in area or delay. Delay time is in pico-seconds (ps).

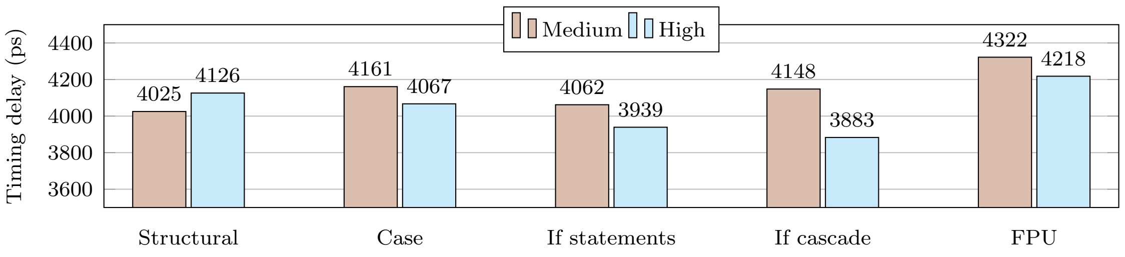


Figure 3.11: Worst path delay of each design style.

Observe that when considering the function unit sharing, the designers tend mainly to think on reduction of the function unit (Cell) area, and not on the reduction of the interconnection (Net) area. However, due to the interconnect scalability problems and their dominating influence on the design in modern nano-CMOS technologies, it could be observed larger changes in the interconnect area than in the cell area.

Also, observe that the RTL compiler optimization techniques involving resource sharing may cause very high circuit area changes (as high as -68% and +63% in the performed experiments), and involve substantial area/delay tradeoffs. Since for the modern nano-CMOS circuit implementation technologies the circuit power consumption of various circuits implementing a given computation is roughly proportional to the circuit area [9], these optimization techniques also may cause very high circuit power consumption changes and substantial power/delay tradeoffs.

3.2 Proposed hardware sharing framework extension

In this Section, the datapath merging problem is presented as an graph-based matching problem, based on the datapath merging method presented in [28, 29]. The datapath merging is an extension to the framework proposed in Chapter 2, as shown in Fig. 3.12. Basically, the merging process takes two DAGs at a time, representing the datapaths of two identified complex instructions, e.g. G_1 and G_2, and merges them into a new combined DAG G_M, representing the combined datapath of the former complex instructions, that overlaps the identified equivalent operations and

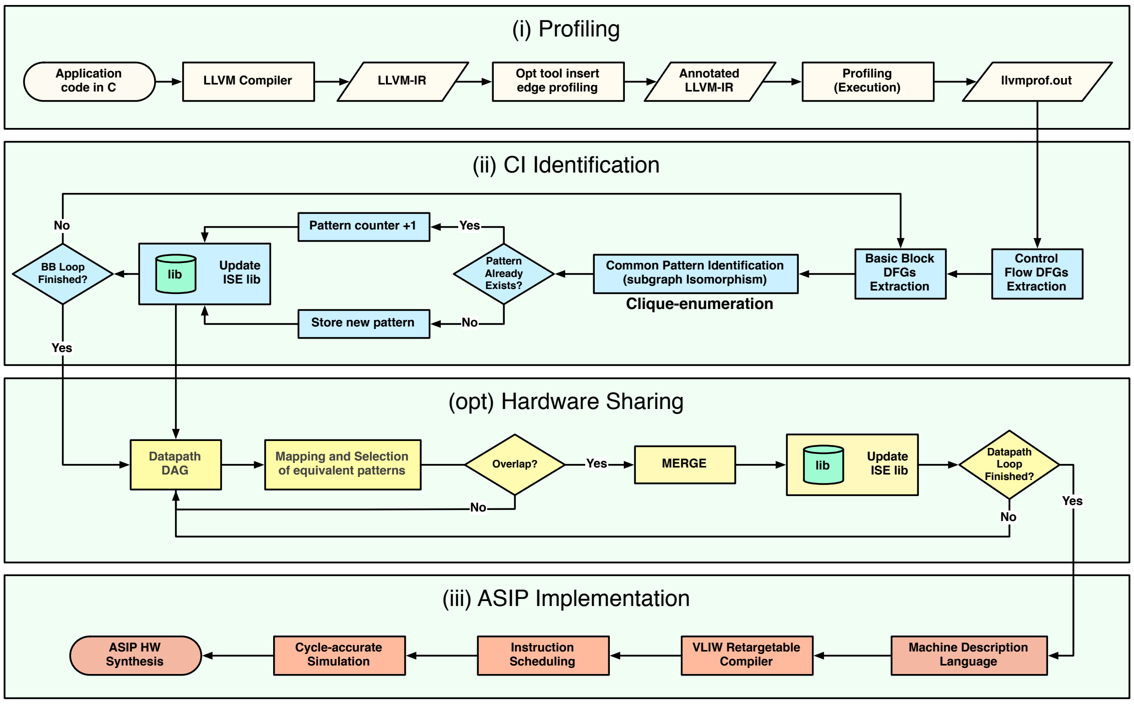
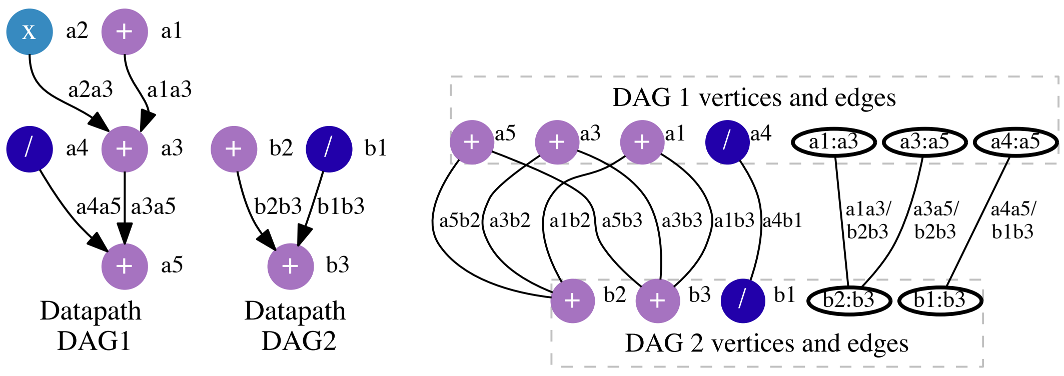


Figure 3.12: Instruction Set Customization Design Flow with a Hardware Sharing optimization stage.

edges of G_1 and G_2. Thereafter, the new merged graph G_M is again merged with the next available graph, e.g. G_3, etc. This process is repeated until the datapath DAG of all the identified complex instructions have been completely or partially fused into one or more datapaths.

3.2.1 Compatibility graph hardware sharing

The datapath merging method based on a compatibility graph is essentially the same that has been used for the custom instruction identification framework. The difference is that the identified subgraphs need to merged to one another, instead of separately stored into a library of complex instructions. The compatibility graph [28, 37] is a suitable method to organize compatible matchings, a.k.a mappings, into a graph representation. The compatibility graph, as previously defined in Section 2.2.1.2, is an undirected graph $G_c = (V_c, E_c)$. Each vertex $v \in V_c$ of the compatibility graph represents a mapping of equivalent patterns (vertices or edges) identified during the matching stage. For instance, let DAG_1 and DAG_2 be two directed acyclic graphs, each representing a datapath, as shown in Fig. 3.13(a). From the bipartite matching depicted in Fig. 3.13(b), it can be noted that each matching will become a vertex of the compatibility graph, as depicted in Fig. 3.14(a). Each edge $e \in E_c$ represents a pair of matchings that are compatible to each other, *i.e.*, which are not ambiguous.



(a) Datapath DAGs. (b) Matching between DAG_1 and DAG_2.

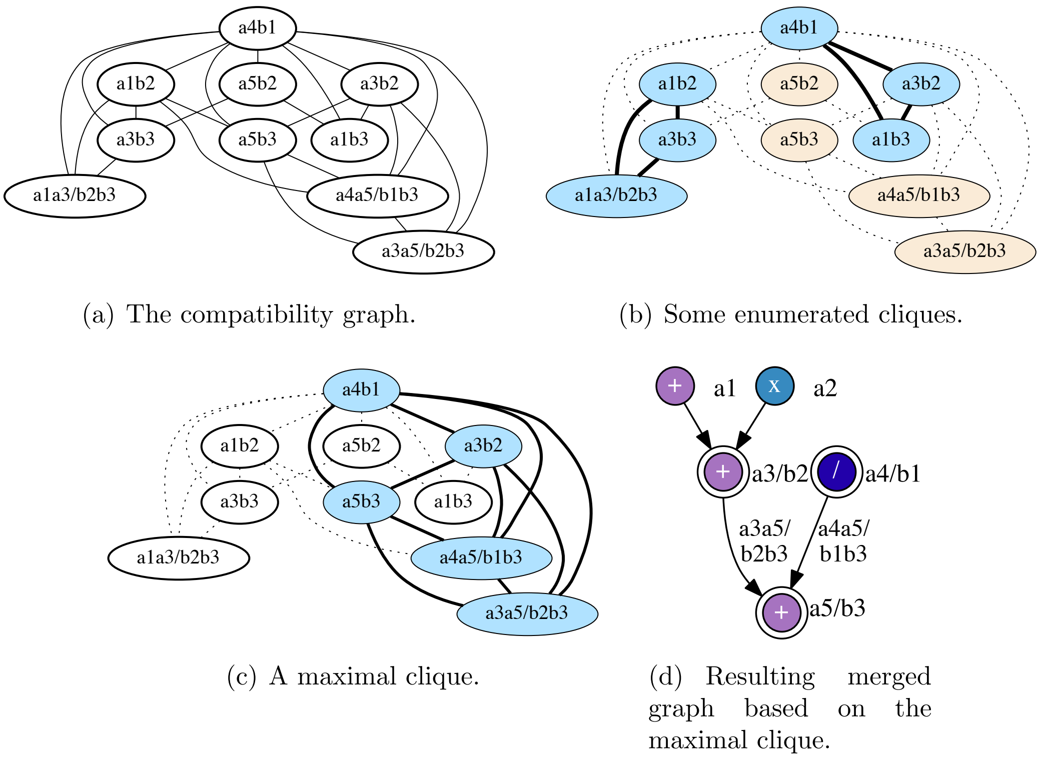
Figure 3.13: Datapaths DAGs and their respective bipartite match.

Observe that each vertex of the compatibility graph is, by itself, a possible merging of identified patterns. For example, vertices a_1b_3, a_4b_1 and a_3b_2 of the compatibility graph represent three possible mergings that can occur: a_1 being merged to b_3, a_4 being merged to b_1 and, finally, a_3 being merged to b_2, because they are all compatible, *i.e.*, they are connected with each other. This is equivalent to identifying a *clique* in the compatibility graph G. Fig. 3.14(b) shows some examples of identified cliques in the compatibility graph, corresponding to the matching of DAG_1 and DAG_2 shown in Fig. 3.13(b).

Identifying maximal cliques produces larger sets of possible mergings that can occur between two graphs (datapaths), thus increasing the hardware sharing potential. Back to the example of Fig. 3.14(a), vertices a_4b_1, a_3b_2, a_5b_3, a_4a_5/b_1b_3 and a_3a_5/b_2b_3 form a maximal clique, as illustrated in Fig. 3.14(c), and the merged graph for the given maximal clique is shown in Fig. 3.14(d).

3.2.2 Experimental results on pseudorandom graphs

For the experiments reported in this Section, the state-of-the-art datapath merging methods presented in Section 1.3 were implemented, executed and compared to the maximal-clique enumeration datapath merging method (CLIQUE). These are: bipartite matching (BIP), longest common subsequence (L-SEQ) and longest common substring (L-STR). A pseudorandom task graph generator [59] is used to produce a set of directed acyclic graphs to generate the sets of custom instructions and their corresponding datapath that are going to be merged. Thus, two groups of experiments were generated. In the first group, ten batches of DAGs were generated, each batch containing ten DAGs to be merged. Each DAG vertex can have up to 2 input connections and 3 output connections. In the second group, ten batches of DAGs



(a) The compatibility graph.

(b) Some enumerated cliques.

(c) A maximal clique.

(d) Resulting merged graph based on the maximal clique.

Figure 3.14: Compatibility graph of DAG_1 and DAG_2, together with a merging example obtained from a maximal clique.

were generated as well. However, each batch of the second group contains fifteen DAGs to be merged. Furthermore, the pseudorandom graph generator has been configured with a probability of generating more similar vertices and interconnections in the second group of experiments. The circuit area and energy estimation of each DAG vertex, *i.e.*, function unit, follows that presented in Section 2.3.2.

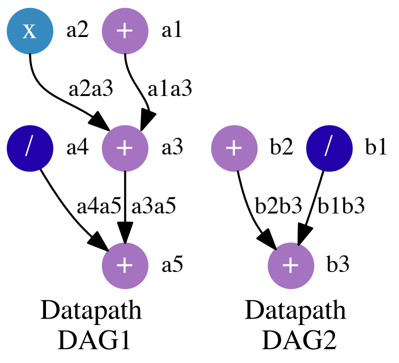
Moreover, the bipartite matching, longest common subsequence and longest common substring methods have been modified to implement up to 2 levels of interconnects optimization. That is because after the mapping of vertices, the bipartite graph may contain *ambiguous* mappings, *i.e.*, two or more vertices from DAG_1 being matched to one equivalent vertex of DAG_2, or vice-versa. This means that the patterns involved in ambiguous mapping cannot actually be merged, otherwise two vertices that are data-dependent can be fused into a single vertex, resulting in an inconsistence. Thus, the optimization options are the following:

opt 0 Any given set of matchings can be selected, as long as they are not ambiguous.

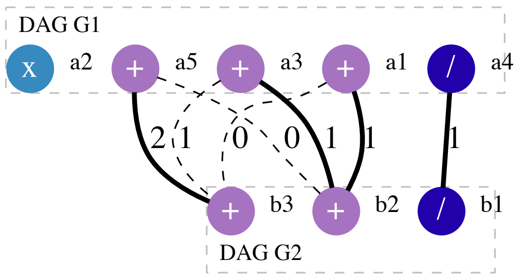
opt 1 The neighbors of the vertices in DAG_1 and DAG_2 belonging to a given match are also compared. Thus, a (+1) weight is added to the corresponding matchings for each equivalent neighbors.

opt 2 A second neighborhood level of the vertices in DAG_1 and DAG_2 belonging to a given match are compared. Thus, a (+1) weight is added to the corresponding matchings for each equivalent neighbors.

Fig. 3.15 gives an example of such a method to solve ambiguities, when having as input the bipartite graph of Fig. 3.13(b). In this way, there is a higher potential for merging several vertices in the same path, leaving their interconnects (edges) unchanged. The maximal clique method does not implement such optimization levels, because the matching of vertices and the matching of edges are both analyzed and included in the compatibility graph for later extraction of cliques.



(a) Datapath DAGs.



(b) Weighted matching between DAG_1 and DAG_2.

Figure 3.15: Assignment of weights to the edges of bipartite match.

From the experimental results for active area reduction shown in Table 3.10, it becomes clear that the active area is significantly reduced for each optimization level, with the maximum reductions up to 59.17% and 71.93% for the first and the second group of DAGs, respectively. Also, matching equivalent vertices according to the longest common subsequence (L-SEQ) or substring (L-STR) method provided better area reduction results, on average. Since the bipartite vertex mapping maps all the equivalent vertices that exist between two given DAGs, which produces many possible combinations, this complicates the *selection* process of a good mapping: the one that may provide less interconnections and less active area after the merging process. Also, the path-based mapping usually maps the sequences or substrings of vertices with larger active area, which can contribute to reduce the overall area at the end of the whole process. The maximal-clique method (CLIQUE) produces a reasonable active area reduction based on a single optimization level, because its mapping selection strategy does not depend on the selection of the highest weighted edges of the bipartite mapping.

Using interconnects preservation optimization (*opt 1* and *opt 2*) produces, in most cases, lower active area datapaths and less interconnections. The reason for such active area reduction through interconnection analysis is that the chance of

Table 3.10: Active area reduction.

		Group1				Group2			
Opt.		BIP	L-SEQ	L-STR	CLIQUE	BIP	L-SEQ	L-STR	CLIQUE
0	Avg.	36.61%	38.46%	37.53%	39.20%	44.42%	46.49%	45.95%	43.09%
	Max	45.49%	51.78%	52.34%	52.18%	61.87%	56.98%	53.52%	59.35%
	Min	23.22%	25.29%	25.13%	31.09%	32.48%	34.68%	40.42%	34.84%
1	Avg.	46.87%	48.76%	48.05%	-	49.89%	51.01%	61.72%	-
	Max	58.91%	54.24%	52.73%	-	57.91%	59.10%	71.93%	-
	Min	35.79%	38.71%	40.20%	-	44.26%	39.81%	55.55%	-
2	Avg.	46.25%	48.71%	49.03%	-	54.18%	56.85%	60.85%	-
	Max	59.17%	56.97%	55.39%	-	63.29%	68.19%	70.87%	-
	Min	35.83%	38.00%	35.99%	-	45.17%	46.92%	55.24%	-

merging two particular vertices increases as more neighbors they have in common. In this way, the merging process is capable of selecting better sets of vertices to be merged: those with more neighbors in common. The experimental results for interconnection (edge) reduction are summarized in Table 3.11. It is clear that the number of interconnections is reduced with each optimization level, as well as the active area, except for a few batches of DAGs. The maximal-clique method (CLIQUE) produced, on average, the best reduction of interconnections based on a single optimization level. The main reason for such high interconnection reduction is that the clique method only maps equivalent edges instead of vertices, which tends to keep the edges intact in the merged graph. This is not a limitation of the clique-based method, but a performance optimization strategy adopted in this work to avoid the overgrowth of the compatibility graph with the addition of vertex mappings. The bipartite vertex mapping (BIP) provided, the second best interconnection reduction, probably because this mapping method is not so focused on the active area reduction like the others (L-SEQ and L-STR). Nevertheless, mapping sequences or substrings of vertices also provided substantial reductions of interconnects.

Table 3.11: Interconnection reduction.

		Group1				Group2			
Opt.		BIP	L-SEQ	L-STR	CLIQUE	BIP	L-SEQ	L-STR	CLIQUE
0	Avg.	8.46%	7.73%	6.76%	28.28%	7.46%	6.16%	4.46%	21.69%
	Max	20.59%	15.38%	14.71%	36.92%	13.35%	8.76%	6.19%	35.57%
	Min	1.56%	1.47%	1.69%	20.93%	0%	2.03%	2.77%	17.39%
1	Avg.	21.54%	18.66%	19.53%	-	15.68%	15.21%	15.85%	-
	Max	30.51%	23.53%	26.92%	-	19.80%	20.10%	20.69%	-
	Min	14.58%	13.21%	13.21%	-	10.42%	12.59%	11.79%	-
2	Avg.	22.08%	20.05%	19.13%	-	20.12%	17.07%	17.77%	-
	Max	37.74%	28.13%	23.73%	-	26.29%	23.27%	22.17%	-
	Min	14.71%	6.15%	13.21%	-	13.70%	11.16%	10.48%	-

Finally, the maximum depth of the combined DAG is used as a reference to approximately measure the latency of the final merged datapath. Such maximum depth represents the longest path from a *source* vertex to a *sink* vertex of the DAG. In general, datapath merging may produce longer datapaths, depending on the vertices that are merged during the process. These methods produced datapaths that have on average 45.51% to 65.55% greater depth, as shown in Table 3.12.

Table 3.12: Maximum depth increase.

		Group1				Group2			
Opt.		BIP	L-SEQ	L-STR	CLIQUE	BIP	L-SEQ	L-STR	CLIQUE
0	Avg.	45.51%	32.13%	41.11%	36.34%	63.33%	65.55%	61.92%	43.09%
	Max	58.82%	64.71%	68.75%	50.00%	72.00%	72.00%	75.00%	59.35%
	Min	25.00%	0%	11.11%	16.67%	52.94%	59.09%	47.06%	34.84%
1	Avg.	45.05%	36.85%	41.92%	-	56.63%	58.35%	59.74%	-
	Max	63.16%	53.33%	61.54%	-	71.43%	75.76%	66.67%	-
	Min	25.00%	22.22%	11.11%	-	40.00%	47.06%	52.63%	-
2	Avg.	44.01%	39.05%	42.20%	-	55.25%	60.16%	59.69%	-
	Max	63.16%	53.85%	64.29%	-	66.67%	73.33%	73.08%	-
	Min	14.29%	22.22%	11.11%	-	40.00%	42.86%	52.63%	-

From these results it is possible to observe that exploitation of datapath merging can influence the area of ASIP and accelerator designs to a very high degree and may result in substantial area and interconnects reduction. Thus, hardware sharing techniques involving datapath merging are of high importance for the multi-objective ASIP and accelerator optimization and adequate tradeoff exploitation. Due to the interconnect scalability problems and their dominating influence on the designs in the modern nano-CMOS technologies, the reduction of interconnects is becoming more important. The presented datapath merging methods are capable of saving as much as up to 60% of the datapath active area and up to 30% of the datapath interconnections, at the cost of increasing the latency of the datapath.

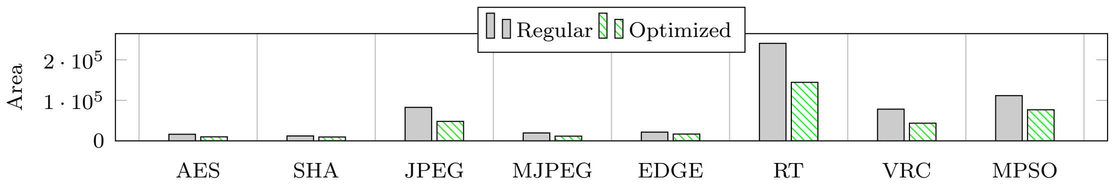
3.2.3 Experimental results on custom instructions

For the experiments reported in this Section, the total circuit area of the library of custom instructions is minimized using the maximal-clique enumeration datapath merging technique [28], which obtained, on average, the most promising results regarding interconnection reduction, together with substantial active area savings and reasonable increase of datapath latency, as presented in the previous Section. The library of custom instructions, available in Appendix A, has been produced by the custom instruction identification framework proposed in Chapter 2. The merged results are available in Appendix B.

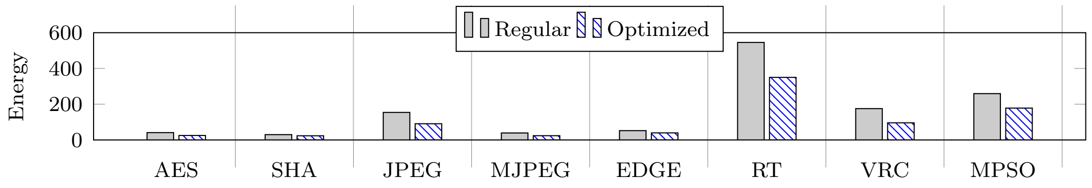
From the experimental results shown in Table 3.13 and in Fig. 3.16, it becomes clear that datapath merging is an important optimization for reducing the circuit area and the energy consumption in the context of instruction set customization. It can be noted that, on average, the circuit area and the energy consumption are reduced by 30%, whereas the number of interconnections is reduced by 6.8%, on average, using the maximal-clique enumeration strategy. Thus, using hardware sharing

Table 3.13: Reduction of total circuit area, static energy consumption and interconnects.

	Area (%)	Static energy (%)	Interconnects (%)
AES	38.81	40.22	10.14
SHA	21.47	21.50	4.26
JPEG	41.90	41.12	10.13
MJPEG	40.72	38.95	6.06
EDGE	22.40	24.06	4.00
RT	35.88	35.83	9.68
VRC	44.28	45.31	10.64
MPSO	31.22	31.27	6.32
Average	30.74	30.92	6.80



(a) Circuit-area reduction.



(b) Static energy consumption reduction.

Figure 3.16: Custom instructions circuit area and energy consumption reduction. The CRC application is not shown because only one complex instruction was identified for it.

may save enough circuit area and energy to enable the replication of hardware accelerators, such as co-processors, or the replication of augmented issue-slots of an extensible VLIW-ASIP, which can further help to improve the performance of an application or class of applications, as will be shown in Section 4.2.2.3.

Chapter 4

Efficient MPSoC with complex instructions speedup

This chapter proposes custom multimedia parallel architectures based on multiple ASIPs with complex instruction speedup, which are usually employed to speedup computing-intensive kernels of highly-demanding applications or class of applications. It also presents a discussion on the state-of-the-art hardware accelerators for well-known 3-D high-fidelity image rendering applications, such as Ray-Tracing and Volume Ray-Casting.

4.1 Custom parallel architectures and accelerators

This Section proposes two different custom parallel architectures [60, 61] for computer graphics applications, ray-tracing and volume ray-casting, respectively. These two applications are capable of producing very high quality image representations of 3-D datasets, but achieving real-time performance is often difficult. Fortunately, both applications have a high parallelization potential, which can be explored to speedup their execution times. Despite that, parallelization alone cannot deliver real-time performance. Thus, a custom parallel architecture may probably be able to speedup the application, especially if many performance-tuned ASIPs, with lower circuit-area requirements and energy consumption, can work together to compute the applications most frequently executed kernels, in parallel.

4.1.1 The GridRT macro-architecture

The GridRT architecture [62, 63] is a massively parallel approach to intersection checks in ray tracing. It is inspired in the Uniform Grid spatial subdivision of the scene [64], which splits the 3-D scenario into 3-D regions of equal size, known as *voxels*. Each voxel contains a list of the 3-D objects that are inside or partially inside the voxel boundaries, as depicted in Fig. 4.1. Usually, an 3-D object is composed of several triangles [65]. Only those voxels that are pierced by a given ray have their objects (triangles) tested for intersections, greatly reducing the number of intersection checks. Also, once an intersection is determined, no further voxels need to be visited for the given ray, since every other intersection cannot be smaller than the given one.

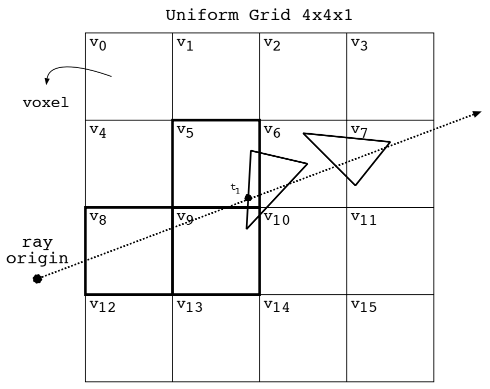


Figure 4.1: The Uniform Grid sequential traversal.

The standard uniform grid intersection algorithm proceeds sequentially, starting the search for intersections from the voxel that is closest to the ray origin to the furthest voxel, until an intersection is found or until the furthest voxel is reached without any results, as in Fig. 4.1. On the other hand, the GridRT macro-architecture maps each voxel onto a Processing Element (PE), responsible for computing intersection checks within its list of scene objects. Thus, all the PEs that are pierced by a ray are going to compute intersections in parallel along the same ray, as in Fig. 4.2. For that reason, it is necessary to discover which PE holds the result that is closest to the given ray origin.

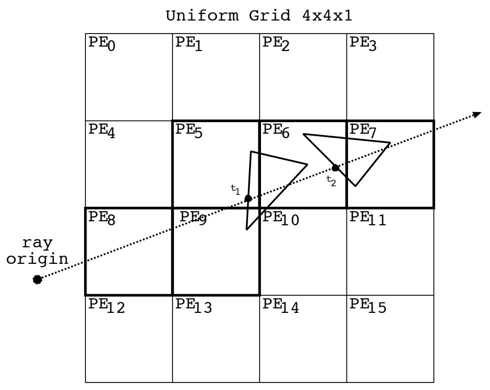


Figure 4.2: Parallel intersection checks.

One naive solution is to exchange the results between every PE that has been processing the same ray. This solution would require every PE to synchronize and, hence, wait for the others to finish their computation until they could exchange their results. Instead, the GridRT architecture uses the traversal order that is inherited from the uniform grid traversal algorithm to determine the closest result. Therefore, every PE is aware of its position, based on the traversal list for a given ray. For example, in Fig. 4.2, the traversal list is L(8,9,5,6,7), from the closest to the furthest voxel (PE). Notice that each ray produces its own list of traversal, although some rays may traverse the same set of PEs. So, based on its position in the list, a PE can take one of the following actions:

- First in the list: if the first PE in the list finds an intersection, every successive PE in the list can abort its own computation. Thus, the first PE sends an interrupt message to the following in the list, which then aborts its computation and forwards the message to the next PE in the list, until the last PE is reached.
- Last in the list: if the last PE in the list finds an intersection, it must always wait for the previous PEs in the list to finish their computation before it can assume to have the closest result. Thus, the last PE must wait for a feedback message from the previous PE or an interrupt message as well, in order to take a decision upon its result.
- Middle of the list: if a PE that is located in the middle of the list finds an intersection, then it can send an interrupt message to the following PEs in the list, but must also wait for a feedback or interrupt message from the previous PE, before it can assume to have the closest result.

At the end of the computation of one ray, only one PE will remain active, while all the others either have finished their computation without obtaining any contributing result or have been aborted. In that way, it is no longer necessary to wait for every PE to finish the computation and exchange results. Parallel intersection checks can be performed and the result correctness is guaranteed. Moreover, the macro-architecture is not restricted to a specific target micro-architecture implementation of PE computations and, hence, can be mapped to a broader range of many-core architectures, such as application-specific instruction set processors (ASIPs) or GPUs, as it will be shown in Section 4.1.1.2.

4.1.1.1 GridRT with ASIP

To implement GridRT architecture presented in the previous Section, each PE can be implemented as a simple and very efficient ASIP, specified in hardware description language (e.g. VHDL). Thus, the datapath of the PE depicted in Fig. 4.3 is controlled by a straightforward controller and every PE is responsible for computing intersection checks within its piece of scene data, which is stored into the *Scene Memory*. The *Control Memory*, where the microprogram is stored, has access to a dedicated register file. Such register file is available for storing intermediate results of a special instruction dedicated to ray-triangle intersection checks.

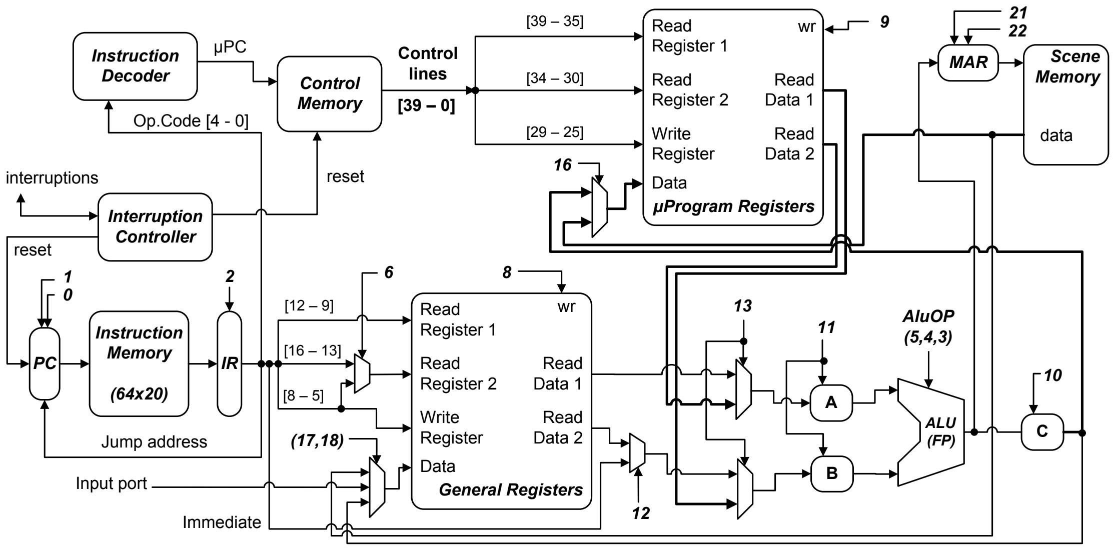


Figure 4.3: The Processing Element processor. Each number is a circuit-level operation.

Moreover, every PE is connected to its direct neighbors through two interrupt lines. The first one is dedicated to the interrupt signals while the second to the feedback signals, as described in Section 4.1.1. The path of both interrupt signals is selected for every new ray, based on its traversal list. Once all PEs are connected, the group of PEs acts as a massively parallel ASIP intersection co-processor for ray tracing, as depicted in Fig. 4.4 for a co-processor example with 4 PEs. Thus, the co-processor receives a ray together with its corresponding list of traversal and activation signals for each PE. Then, when the ray-triangle intersections have finished, the co-processor returns the closest result to the given ray origin.

The GridRT co-processor can be controller by any main processor. In this case it is a MicroBlaze RISC microprocessor IP from Xilinx [66], which can be synthesized in FPGAs, as shown in Fig. 4.5. Such connection is made through a Fast Simplex Link (FSL) channel [67], a low-latency point-to-point communication link available in MicroBlaze. Therefore, the MicroBlaze microprocessor executes the ray generation and the ray traversal algorithms, providing the necessary input data to the co-processor. The result for each ray is then read from the co-processor and transmitted to a host processor via a UART interface [68], for post-processing and visualization.

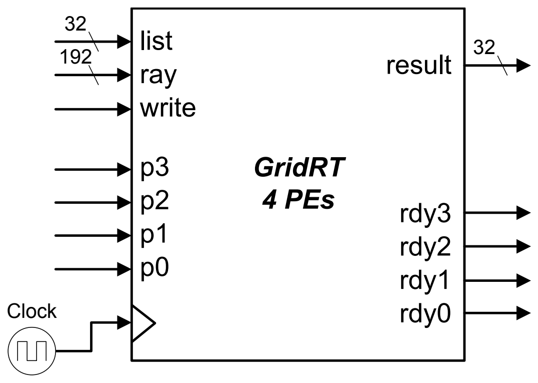


Figure 4.4: The GridRT co-processor, with 4 PEs.

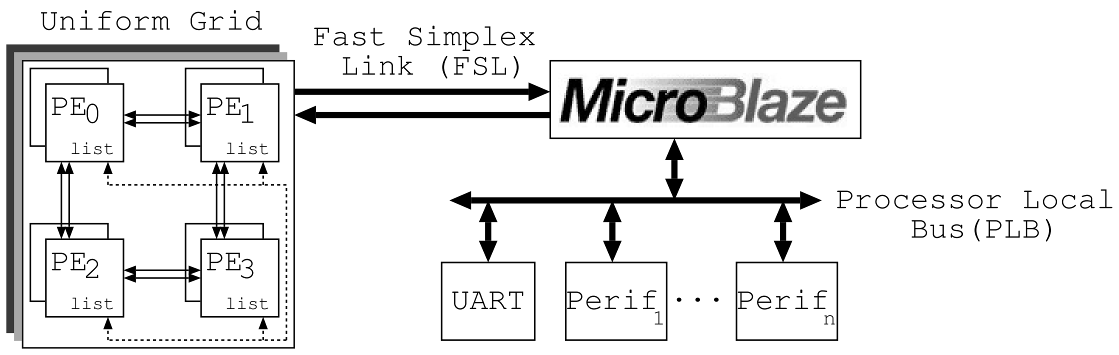


Figure 4.5: The MicroBlaze microprocessor connected to a GridRT co-processor.

Following the parallel model of the GridRT architecture in Section 4.1.1, parallelism is achieved through parallel intersection checks and exchange of signaling messages to determine the correct result. Thus, each PE is connected with its direct neighbors by two interruption lines. Signals are handled by a Interrupt Controller present in every processing element, as highlighted in Fig. 4.3. The first interrupt informs the current PE that a previous one has already computed an intersection and, since no further intersection can be closer due to the traversal list order, the PE forwards the interruption and aborts its computation. The second interrupt deals with the situation when a PE has found an intersection, but has not received any feedback from a previous one. Thus, the second interruption signal informs the current PE that every former processor has finished the computation without results. At the end, only one remaining result prevails, while others had finished or were aborted. State machines are responsible for detecting an intersection result within the PE and building the interrupt signals path, as depicted in Fig.4.6. Simple multiplexers are used to select the interruption path based on the traversal list received together with the ray data. Thus, each PE knows what is the next PE and the previous PE according to its position in the list.

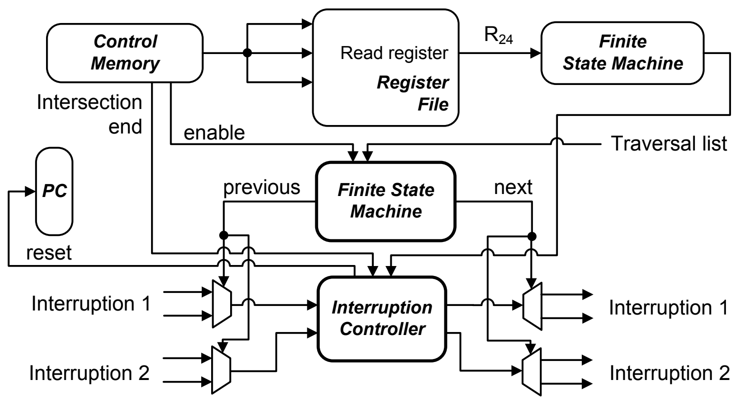


Figure 4.6: Interrupt Controller detailed datapath.

4.1.1.2 GridRT in GPU

While the GridRT implementation in ASIPs maps each PE onto an ASIP, the counterpart GPU implementation maps each PE onto a Block of threads, that in turn is organized as a Grid of Blocks, according to the Compute Unified Device Architecture (CUDA) [69]. Such CUDA architecture model aims at performing a massive number of floating-point calculations simultaneously. Thus, it can be used across a wide range of applications that can be parallelized under the CUDA programming model. Here, the goal is to take advantage of the CUDA paradigm to implement the GridRT parallel model described in Section 4.1.1.

In the CUDA programming model, all threads in a grid execute the same *kernel* function. Thus, each thread is assigned a unique identifier to distinguish it from others. Besides, groups of threads are organized into blocks and, hence, have access to a fast local *shared memory* and can be synchronized using a *barrier synchronization* function. On the other hand, threads in different blocks cannot be synchronized via barriers. Each block is also assigned a unique identifier. In modern GPUs, depending on the configuration that is specified when a kernel function is launched, each block or thread identifier can have up to three dimensions (x, y, z). For example, if the data to be processed is organized as a matrix $M(x, y)$, threads can be organized in two dimensions $(ThreadIdx.x, ThreadIdx.y)$, so that they can be easily assigned to its corresponding matrix data.

At the architectural level of current generation of hardware, blocks of threads are assigned to Streaming Multiprocessors (SMs), each one consisting of up to 32 CUDA Cores, as shown in Fig. 4.7. Once a block is assigned to a SM, it is split into *Warps*, which are groups of 32 threads with consecutive identifiers. Each block can include up to 1024 threads. A *Warp* is scheduled for execution by a *Warp Scheduler*. Thus, if an instruction, say i, that is being executed is waiting for a previous one whose completion is delayed due to a required long-latency operation, then a different *Warp* may be selected for execution of instruction *i*. In that manner, the resources of an SM are better exploited and this is called *latency hiding*.

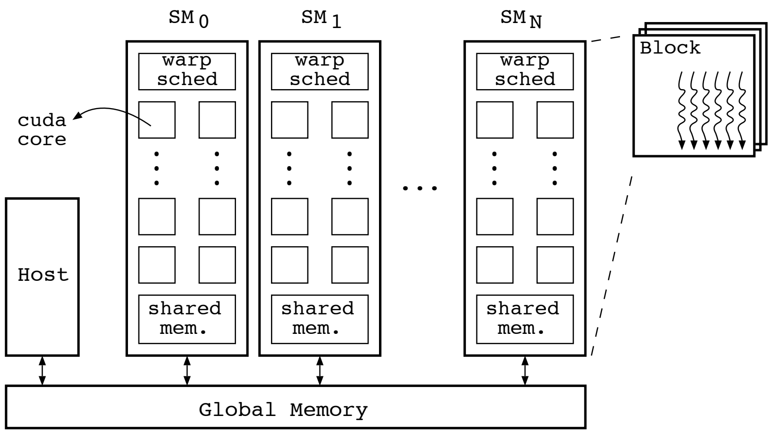


Figure 4.7: Streaming Multiprocessors (SMs) organization, with each SM executing a blocks of threads.

The GridRT implementation in CUDA maps a PE onto a block of threads, as depicted in Fig. 4.8. However, the threads of different blocks cannot coordinate theirs activities. Therefore, a different, yet similar, approach from the ones presented in Section 4.1.1 must be used to determine the result that is closest to the ray origin.

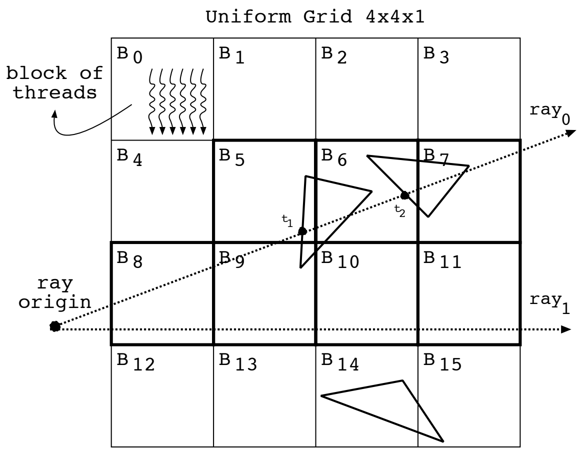


Figure 4.8: GridRT–CUDA configuration.

A possible solution is to map each PE onto a thread, instead of blocks of threads. Hence, all the threads in charge of processing a ray could store their results in the shared memory and thereafter synchronize their work at the end of the given ray processing, checking which result is the closest to the ray origin. However, this solution has a major drawback: in order to make usage of the shared memory, only one block of threads, representing the whole GridRT, could be executed. All the other blocks would become useless, causing a waste of too many valuable resources.

Therefore, a different approach is proposed. Instead of mapping each PE onto a thread, each PE is mapped onto a block of threads and the host processor is in charge of determining the correct result at the end of the whole computation. Now each ray has an array of results associated to it. The size of such an array corresponds to the maximum number of PEs, i.e. blocks, that can be traversed by a given ray. The size is determined by the total number of subdivisions according to each of the three axis (n_x, n_y, n_z) of the GridRT spatial structure, as defined in Eq. 4.1. For instance, considering the grid of Fig. 4.8, the maximum size of the array is $N = 7$, since the uniform grid subdivision is $n_x = 4$, $n_y = 4$ and $n_z = 1$.

$$N = n_x + (n_y - 1) + (n_z - 1) \tag{4.1}$$

When each block of threads has finished the intersection checks with respect to the the corresponding voxel, the result is stored in the array at the block associated entry. Thereafter, the block can proceed with the computation of a different ray, which also has a different array of results associated to it. In the end, the host processor copies the matrix of results from the GPU, wherein each row corresponds to the array of results computed by the block for a given ray. Considering once again the example in Fig. 4.8, the matrix is shown in Table 4.1. Furthermore, each column contains the result that was computed by a block, also according to the list of traversal associated to each ray.

Table 4.1: Matrix of results copied from the GPU by the host processor.

Ray	Array	Index						
Ray_0	$List_0$	B_8	B_9	B_5	B_6	B_7	-	-
	$Result_0$	-	-	t_1	t_2	-	-	-
Ray_1	$List_1$	B_8	B_9	B_{10}	B_{11}	-	-	-
	$Result_1$	-	-	-	-	-	-	-
			⋮					
Ray_n	$List_n$	B_k	B_{k+1}	B_{k+2}	-	-	-	-
	$Result_n$	-	t_i	t_{i+1}	-	-	-	-

Once all the intersections checks have been computed, the host processor checks each ray of the matrix, searching for the nearest intersection results. Once an intersection is encountered, it proceeds to the next ray, since the order of traversal guarantees that the encountered result is the closest to the ray origin. After processing all the results, the host creates a new set of rays based on the intersection points found by the previous set of rays. Thus, several calls to the GridRT *function kernel* produce a new set of intersection points for *secondary rays* and *shadow rays*. In the end, the results are processed by the host processor in order to created the final image from the ray traced 3-D scene, with shadows, reflections and refraction effects.

It is important to note that every block of threads is not only responsible of parallel processing of rays, but also parallel intersection checks within the block. Depending on the number of blocks and threads per block that are set at the kernel execution, one block may execute up to 1024 threads, split into *Warps* of 32 threads. Thus, each thread can be assigned to process one ray-triangle intersection in parallel with others, only if the number of triangles that belongs to the block is also compatible. In general, the more subdivisions are applied to the uniform grid spatial structure, which means a higher granularity of voxels (Blocks or PEs), the less triangles are present per voxel. In consequence, it became possible to assign a thread to each ray-triangle intersection check. Taking advantage of the fast access *shared memory* within a block, the intersection results are then stored in this memory and, in the end, the closest result to the ray origin in respect to the given block is stored in the matrix, as described in Table 4.1 for the grid of Fig. 4.8.

4.1.1.3 Experimental results

The GridRT–FPGA hardware architecture based on ASIPs was described in VHDL, while the GPGPU version was described in CUDA v4.0. The hardware architecture was simulated in ModelSim XE 6.3c and synthesized to a Xilinx Virtex-5 XCE5VLX50T FPGA, through Xilinx ISE Design Suite 11.1. Initially, the GPGPU version was running in one NVidia GTX 470 GPU. Later, a second GTX 470 GPU was included, working in parallel with the first one. In such dual-GPU configuration, the same kernel is running in both GPUs, but with the input primary rays split between them. In that way, an even higher level of parallelism is achieved.

First of all, the area cost of the ASIP-based implementation is presented in Table 4.2, for a GridRT co-processor of 1, 2, 4 and 8 PEs, respectively, including the MicroBlaze microprocessor and its peripherals.

Table 4.2: Area cost for 1, 2, 4 and 8 PEs, in contrast with old vs. new GridRT architecture.

Resources	GridRT–FPGA			
	1	2	4	8
Slice Reg	5012	7578	12368	21964
Slice LUT	6168	9254	15521	28138
LUT-FF	3184	3828	6851	14866
BlockRAM	31	32	48	80
DSP48Es	9	13	21	37

Synthesis Target: Virtex-5 XCE5VLX50T.

Parallel execution of up to 8 PEs was possible on a single Virtex-5 FPGA, running only at 50MHz. Each PE is equipped with four floating point units from Xilinx IP catalog [70], delivering an average of 10 MFLOPs per Processing Element. Each intersection check takes 307 cycles, involving 58 floating point operations. If all 8 PEs are executing in parallel, an estimated peak of 80 MFLOPs can be achieved.

On the other hand, the Graphics Processing Unit (Nvidia GTX 470) used to execute the GridRT kernel operates at 1.22Ghz, which is almost 25 times faster than the FPGA hardware GridRT implementation, due to the clock cycle limitations of the FPGA board. Also, such GPU contains up to 448 CUDA cores, each one equipped with at least a floating point unit, which is 14 times more floating point units than those available in all 8 PEs of the GridRT dedicated hardware altogether. Furthermore, if each fused multiply-add (FMA) operation is considered as two floating point units, then the GPU would contain 28 times more floating point units. The execution times for primary rays processing are presented in Table 4.3, from 1 to 8 PEs and Blocks, for the dedicated GridRT hardware and for the GridRT kernel in GPU, with the latter using one and two GPUs, respectively. The 3-D scene that was rendered is a low-polygon count of the Stanford Bunny (Low-res Stanford Bunny 3-D scene in Fig. 4.9(a)), for a resolution of 320 $\times$ 240 [71] only, also because of the FPGA area and memory constraints. In Table 4.3, all times are given in seconds.

Table 4.3: Dedicated hardware and GPGPU kernel execution times.

Architecture	Number of PEs and Blocks			
	1	2	4	8
GridRT–FPGA	337	184	130	82
GridRT–CUDA (x1 GPU)	6.77	3.74	1.86	0.99
GridRT–CUDA (x2 GPU)	3.03	1.79	0.71	0.29

*All times are in seconds. Low-res Stanford Bunny 3-D scene.

From Table 4.3, it is possible to observe that as more processing elements (or blocks of threads) are added to the architecture, the rendering time is almost linearly reduced. The GPGPU implementation running in one GPU is up to 82 times faster

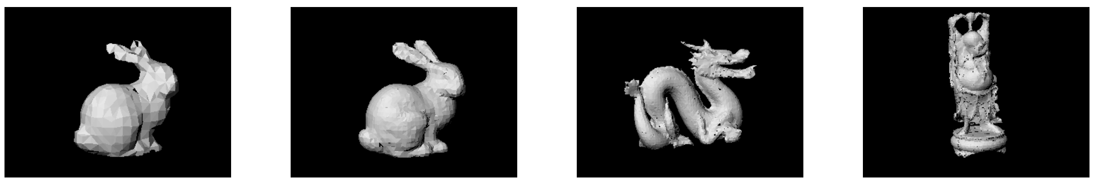

(a) 948 triangles (b) 3851 triangles (c) 11102 triangles (d) 15536 triangles

Figure 4.9: 3-D Scenes, rendered with primary rays only in GPGPU.

Table 4.4: GridRT–CUDA kernel execution times.

3-D Scene, 1 GPU	Blocks of threads											
	1	2	4	8	12	18	27	36	48	64	125	216
Bunny low-res	6.77	3.74	1.86	0.99	1.23	1.20	1.31	1.34	1.70	2.47	5.00	8.60
Bunny hi-res	-	-	-	1.93	2.52	3.29	3.03	4.3	5.97	6.87	11.42	11.38
Dragon	-	-	-	-	-	-	-	13.3	13.98	18.04	31.79	37.02
Happy Buddah	-	-	-	-	-	-	-	-	17.43	22.44	34.55	48.20

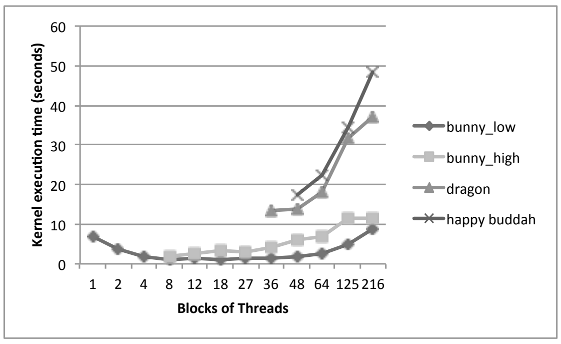
3-D Scene, 2 GPUs	Blocks of threads											
	1	2	4	8	12	18	27	36	48	64	125	216
Bunny low-res	3.03	1.79	0.71	0.29	0.28	0.26	0.29	0.29	0.44	0.68	1.31	2.5
Bunny hi-res	-	-	-	1.3	1.25	1.34	1.52	2.21	2.7	2.9	5.03	10.55
Dragon	-	-	-	-	-	-	-	6.67	6.1	8.02	14.12	17.02
Happy Buddah	-	-	-	-	-	-	-	-	7.80	10.03	15.10	22.17

than the ASIP-based FPGA implementation. If two GPUs are running in parallel, the kernel is around 280 times faster then the ASIP-based and only 3.4 times faster than using only one GPU. The performance gap between the GPGPU-based and the ASIP-based implementations can be explained by the generality and programmability overhead intrinsic of FPGA technology, resulting in 25 times slower clock than for GPGPU, and by the massive floating-point parallelism provided by the GPGPU, especially if two GPUs are being used, increasing even more the technology gap. If the ASIP-based GridRT would be implemented in ASIC technology, instead of FPGA, comparable to the GPGPU technology, then it could most probably run with more than 25 times faster clock, and be faster than the GPGPU implementation, as its hardware is much simpler as the hardware of GPGPU.

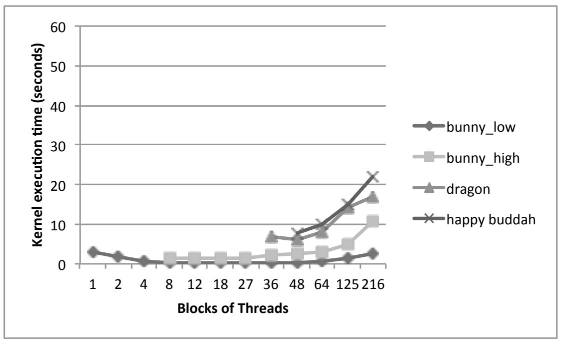
Table 4.4 presents further kernel execution times of up to 216 blocks of threads for the GridRT–CUDA implementation. The execution for some blocks of threads could not be measured because of the GPU architecture limitations. For instance, running the GridRT kernel with only one block of threads means that there is no subdivision of the 3-D scene, so all the triangles belong to only one voxel (block of threads). If there are too many triangles in a block of threads (more than 1024) then the kernel cannot be executed, because the maximum number of threads per block is 1024.

From Table 4.4 it is possible to observe that the GridRT–CUDA achieves acceleration when up to 18 blocks of threads (or Processing Elements) are employed in the low-res bunny 3-D scene. Beyond that, the performance degenerates, as also depicted in Fig. 4.10(a). For all the other 3-D scenes, the performance degenerates faster, because of the higher number of triangles and intersection tests that need to be computed.

The performance degeneration can be explained by the work granularity level that each block of threads is operating at. Parallel intersection checks are performed by all threads within a block, but *loops* and *conditional branches* dominate most part of the computation. It is well-known that such conditional constructions are not well suited for the *Stream Processing* model (Single Instruction Multiple Data,



(a) Kernel execution in 1 GPU.



(b) Kernel execution in 2 GPUs.

Figure 4.10: GridRT–CUDA kernel execution results. Resolution: 320×240 primary rays.

SIMD). Control flow has been for years optimized for *Von Neumann* architectures and are better executed by such. Also, the performance degenerates because more blocks of threads are competing to be executed by the streaming multiprocessors available in the Nvidia GTX 470 GPU.

Using two GPUs, as shown in Table 4.4 and Fig. 4.10(b), less performance degeneration is observed. Also, a dual-GPU configuration is around 2 and 5 times faster, when compared to the kernel execution times in one GPU. The only difference between both implementations is that, in the latter, the primary rays are split for execution among the two GPUs. Such acceleration suggests that parallel execution of rays is at least as efficient as parallel computation of intersection tests.

4.1.1.4 Results overview

Two implementations of the massively parallel GridRT architecture for Ray Tracing are discussed: the ASIP-based FPGA implementation with an embedded intersection instruction and GPGPU (CUDA). These two implementations are analyzed and compared regarding performance. The ASIP-based GridRT implemented in a single Virtex-5 FPGA can execute up to eight processing elements in parallel, running at 50MHz only. When implemented in an ASIC technology, instead of FPGA, it could most probably run more than 25 times faster. As more PEs are included, the better is the performance achieved. However, recent advancements in GPGPU architectures, such as in the Nvidia *Fermi* architecture, have enabled an efficient implementation of several parallel applications, including ray-tracing itself. Therefore, the GridRT architecture has been mapped to GPGPU using CUDA, with minor modifications in the synchronization of results, performed by the host processor. The GPGPU implementation in one GPU with 25 times faster clock achieved 82 times higher performance than the FPGA ASIP-based GridRT, which shows the architecture potential towards real-time ray-tracing. After a second GPU had been included to the system, the GPGPU implementation achieved around 280 times higher performance compared to the ASIP-based implementation.

One of the reasons for such speedup gain is explained by the processing power gap that exists between the FPGA programmable hardware implementation and the GPGPU. For instance, the first is running at only 50MHz and can hold up to eight processing elements. The latter runs at frequencies up to 1.2GHz and can execute many blocks of threads. In total, the Nvidia GTX 470 GPGPU used in the intersection checks have 448 CUDA cores, each one containing at least one floating-point unit. In total 14 times more floating point units than employed in a GridRT FPGA implementation of eight PEs. Moreover, if each fused multiply-add operation yields 2 floating point units, then the GPGPU would have 28 times more floating point units. Thus, if the same implementation technology and processing power was available to the ASIP counterpart implementation, its performance would be comparable to that of GPGPU. The second main reason for the speedup gap is that the ASIP-based GridRT can only perform parallel intersection checks for one ray at a time for now, while the GPGPU implementation can already perform parallel intersection checks for more than one ray.

Summing up, this work demonstrated that the GPGPU implementation of the GridRT macro-architecture for ray-tracing is able to deliver a high performance, 82 times higher than that of the ASIP-based FPGA implementation. However, since the GPGPU implementation introduces more hardware overhead comparing to an ASIP-based ASIC implementation, the ASIP-based ASIC implementation is expected to have lower area and power consumption. For instance, a hypothetical ASIC-based GridRT implementation in 40nm, the same technology as the GTX 470 GPU, and running at 1.2GHz, would yield 227MFLOPs per processing element, which is 22 times faster than the current implementation in FPGA. Furthermore, if each ASIP is considered equivalent to 4 CUDA cores (in terms of employed floating point units), then at least 112 ASIPs could be synthesized in the ASIC technology, yielding around 2.5GFLOPs of throughput. Still, the GTX 470 throughput would be 34 times higher (around 850GFLOPs), but the custom architecture is application-specific and because of that should be able to more efficiently compute the parallel ray-tracing algorithm, and probably consuming less power.

4.1.2 Parallel volume ray-casting MPSoC

High performance visualization of 3-D datasets has always been one of the main goals in Computer Graphics. For 3-D volumetric datasets, such as those acquired by *Computer Tomography* (CT), the rendering process is generally known as Volume rendering. The volumetric dataset is usually composed of several stacked parallel slices (images) that form a 3-D volumetric dataset. There are different techniques to render 3-D volumetric datasets [72, 73]. For instance, the *Marching Cubes* algorithm [74] is one approach to turn voxels samples into polygonal data, in order to create an actual set of 3-D primitives that can be rendered by regular GPUs pipeline. On the other hand, such technique may lead to a poor quality polygonal representation of the volume, because of the approximations that are performed to create the polygonal data. Thus, the Volume Ray-Casting algorithm is a better candidate for producing more accurate results [75, 76]. Essentially, this algorithm samples equidistant points along the ray, inside the volumetric 3-D dataset. Each sample, i.e. Voxel (*volumetric pixel*), corresponds to a given color and opacity in one of the parallel slices of the dataset. The interpolated colors and opacities are merged through *compositing* to yield the color of the view-plane pixel through which a primary ray has been traversed. For instance, the algorithm can show specific parts of a human body volumetric dataset, such as bones or internal organs.

Interactive visualization of volumetric datasets is often difficult. The volume ray-casting performance can drop significantly as more complex datasets are used. On the other hand, volume ray-casting has a very high parallelization potential, as each ray can be processed independently, producing one corresponding pixel information. Therefore, there are consistent approaches to accelerate volume ray-casting with custom parallel architectures in hardware. In [77], a pipelined application-specific integrated circuit (ASIC) was created, fabricated in 0.35 μ technology and running at 125MHz. Such ASIC is capable of producing interactive frame-rates at some degree, since there are limitations regarding the size of the dataset (256^3 voxels). GPUs have recently become a good option for massively parallel processing of floating point data [69]. Thus, there are also approaches to accelerate volume ray casting using GPUs [78, 79]. However, most of the volume ray-casting algorithms on GPU strongly depend on optimizations to achieve real-time rendering performance. For example, using *texture* or *constant* memories of the GPU to store frequently-used data can substantially improve the given algorithm performance, because of their much lower latency [69].

In this Section, the implementations of the interactive, un-optimized and flexible parallel volume ray-casting algorithm with *supersampling* on three different multi-core architectures are discussed: Chip Multiprocessor (CMP), Graphics Processing Unit (GPU) and Multiprocessor System on Chip (MPSoC). The CMP implementation of the algorithm uses OpenMP, while the GPU implementation is CUDA-based. The MPSoC-based implementation on FPGA uses the shared DDR memory for synchronization. It extensively compare performance results of the GPU and OpenMP implementations, showing that the GPU implementation can reach interactive visualization, especially when a multi-GPU configuration is used. It also compared and analyzed the advantages of using multi-GPU configuration over a single-GPU configuration for varying workloads (number of primary rays). Finally, the MPSoC-based implementation on FPGA (Xilinx Virtex-5) shows the portability and scalability of the volume ray-casting algorithm, as several microprocessors (MicroBlaze [66] cores) can be mapped on the FPGA and run the algorithm in parallel. All the implementations have not been optimized to use any special features of the corresponding architectures.

4.1.2.1 Parallel volume ray-casting in CMP

The OpenMP-based parallel volume ray-casting technique is presented in Algorithm 3, where a *for work-sharing construct* is used, that splits the execution of the parallel section among the group of threads. Thus, iterations of the *for loop* are split across the group of threads. Therefore, in Algorithm 3, groups of rays are assigned to groups of threads for execution, leading to parallelization of rays. Each thread has its own private variables (i, j and s) that are used to control the loop iterations assigned to each thread in the beginning of the parallel section. Also, if *supersampling* is enabled, then each ray spawns a given number of neighbor sampling rays (i.e. in the vicinity of the primary ray), that are executed by the same thread. Thus, the color information of each pixel is measured from all the sampling rays, improving the overall quality of the resulting image.

Algorithm 3 Volume Ray-Casting with OpenMP

```
Require: rays, uniform grid structure, 3-D dataset
Ensure: image
 1: # pragma omp parallel for private(i,j,s)
 2: for $i = 0$ to WIDTH do
 3:     for $j = 0$ to HEIGHT do
 4:         color pixel;
 5:         for $s = 0$ to N_SAMPLES do
 6:             ray ry ⇐ get_ray(i,j,s);
 7:             color aux ⇐ intersectGrid(grid, ry, dataset);
 8:             pixel ⇐ pixel + aux;
 9:         pixel ⇐ pixel / N_SAMPLES;
10:         image[i][j] ⇐ pixel ;
```

4.1.2.2 Parallel volume ray-casting in GPU

The CUDA-based parallel volume ray-casting is presented in Algorithm 4. In the CUDA programming model, a thread is actually a *lightweight thread*, because of their simplicity and faster context switching mechanism when compared to regular threads. Throughout this Section, threads in CUDA are referred as *lightweight threads.* In addition, the CUDA-based implementation in Algorithm 4 has not been optimized for GPU execution. For example, the kernel do not make use of *shared* *memory* or *texture memory*, that are usually employed to avoid global memory long latency penalties.

Modern general purpose GPUs are capable of executing many thousands of threads in parallel [69]. Thus, each thread can be assigned to a primary ray that crosses a pixel of the view-plane. The result is that a portion of the final image is going to be produced by a *block of threads* (one pixel per thread). The corresponding *CUDA Kernel* is presented in Algorithm 4, considering that all data transfers between the host and the GPU have been already performed. If *supersampling* is enabled, the thread will execute as many sampling rays as required, as shown in line 5 of Algorithm 4. The sampling rays are addressed in column chunks, as shown in line 6.

Algorithm 4 Volume Ray-Casting CUDA–kernel

```
Require: rays, uniform grid structure, 3-D dataset
Ensure: image
 1: ray ry;
 2: i ⇐ blockDim.x * blockIdx.x + threadIdx.x;
 3: j ⇐ blockDim.y * blockIdx.y + threadIdx.y;
 4: color pixel;
 5: for samples = 0 to N_SAMPLES do
 6:     ry ⇐ rays[i][j+samples];
 7:     color aux ⇐ intersectGrid(uniform grid, ry, dataset);
 8:     pixel ⇐ pixel + aux;
 9: pixel ⇐ pixel / N_SAMPLES;
10: image[i][j] ⇐ c;                              ▷ corresponding pixel color
```

Furthermore, the same kernel shown in Algorithm 4 can be executed by multiple GPUs. However, the input rays are split among the GPUs, increasing even more the parallel processing of rays. In order to use two GPUs, a separate thread must be created to access each GPU, because one thread cannot control both GPUs at the same time. For that reason, OpenMP is used to create two threads, each one controlling one GPU. The same idea can be extended for more than two GPUs, if available. In the end, the results from both GPUs are merged by the host process into one single image.

4.1.2.3 Parallel volume ray-casting in MPSoC

The MPSoC architecture consists of up to four Xilinx MicroBlaze [66] microprocessors running in parallel at 125MHz. They are connected to a shared DDR memory via a Xilinx Multi-Port Memory Controller (MPMC) [80]. One of the microprocessors is connected to a few communication peripherals, to enable input/output data transmission between the MPSoC and a host machine, as well as to enable access to the FPGA's flash memory. Thus, all the microprocessors must wait until the whole 3-D volume data is available for computation.

The parallel volume ray-casting implementation is presented in Algorithm 5, where iterations of the *for loop* are split across the microprocessors, as shown in line 2. Therefore, in Algorithm 5, groups of rays are assigned to different microprocessors, since rays can be processed independently from the others. Each microprocessor knows which data to read and to write, according to its own identification number (CPU_ID= $0, 1, 2$ or 3) and also according to the total number of enabled microprocessors (N_CPU= $1, 2, 3$ or 4), as shown in line 2 of Algorithm 5. Finally, at each loop iteration, an image pixel is produced, as shown in line 9 of Algorithm 5.

Algorithm 5 Volume Ray-Casting with MicroBlaze

```
Require: rays, uniform grid structure, 3-D dataset
Ensure: image
1: for ($i = 0$; $i <$ IMG_WIDTH; i++) do
2:     for $j$ = CPU_ID; $j <$ IMG_HEIGHT; j $\Leftarrow$ j + N_CPU) do
3:         color pixel;
4:         for $s = 0$ to N_SAMPLES do
5:             ray ry $\Leftarrow$ get_ray(i,j,s);
6:             color aux $\Leftarrow$ intersectGrid(grid, ry, dataset);
7:             pixel $\Leftarrow$ pixel + aux;
8:         pixel $\Leftarrow$ pixel / N_SAMPLES;
9:         image[i][j] $\Leftarrow$ pixel ;
```

4.1.2.4 Results overview

In this Section experimental results on different datasets for each multi-core architecture implementation are presented. The CUDA-based implementation was compiled using the CUDA Toolkit 4.0, while the OpenMP-based implementation was compiled in GCC 4.4.4. Up to two NVIDIA GTX 470 GPU were used for execution of the algorithm in CUDA, while a Core i7 960 Intel Multiprocessor (at 3.2 GHz) was used for the algorithm execution in OpenMP. The MPSoC-based architecture was synthesized in Xilinx EDK 13.1 for a Virtex-5 XC5VLX50T FPGA and the parallel algorithm implementation was compiled using MicroBlaze gcc compiler 4.1.2. All the execution time results are measured in seconds and the volumetric dataset, shown in Fig. 4.11, is available in [81].

For each volumetric dataset, the volume ray-casting algorithm was executed for 1280×800 primary rays, producing high-resolution images. In addition, the algorithm was executed with *supersampling* enabled, varying from 1 to 32 sampling rays per pixel. Therefore, up to 32 sampling rays were cast around the region of the primary ray pixel, producing smoother edges in the resulting image. The performance results are summarized in Table 4.5, for the OpenMP and the CUDA based implementations, using 1 and 2 GPUs, respectively. The MPSoC does not support

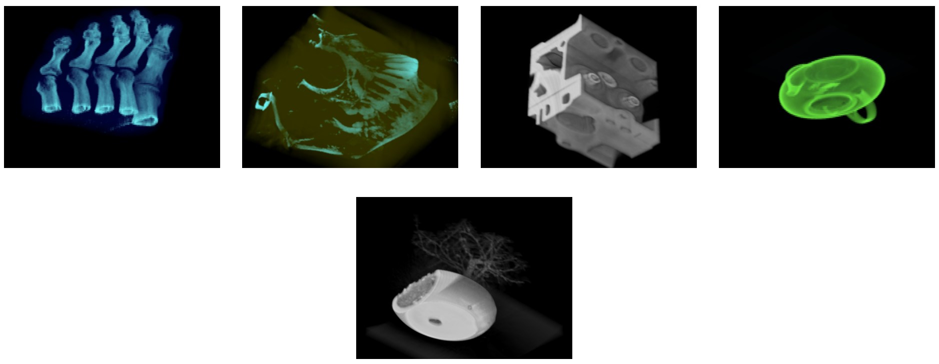


Figure 4.11: Images produced by the proposed parallel volume ray-casting algorithm.

high-resolution volume ray-casting processing because of memory limitations. Thus, its results are not included in Table 4.5.

The OpenMP-based implementation uses 8 parallel threads, since the Core i7 microprocessor can execute up to eight parallel processes. The results in Table 4.5 show that even for one sampling ray, the performance is still not enough to ensure interactive visualization of the datasets. However, good results, i.e. image quality and interactive visualization, can still be obtained at lower resolutions, as fewer primary rays are processed.

On the other hand, the GPU-based implementation results show that interactive visualization of volumetric datasets is possible even for high-resolution volume ray-casting. As depicted in Fig. 4.12(a) and 4.12(b), the volume ray-casting execution time for every dataset is still below one second if up to 4 sampling rays are used, which corresponds to processing $1280 \times 800 \times 4$ rays, in total. Thus, more than one image, a.k.a frame, can be produced in one second, especially if less than four sampling rays are used. The dual-GPU implementation is around 90 times faster than the OpenMP implementation. Comparing the algorithm execution results using one and two GPUs, the performance is almost two times faster when two GPUs are employed instead of one. Also, observe that as more sampling rays are used, the performance gap increases, making the dual-GPU configuration a better candidate for high-quality interactive volume ray-casting, especially for complex datasets such as *aorta* and *backpack*, as shown in Fig. 4.13.

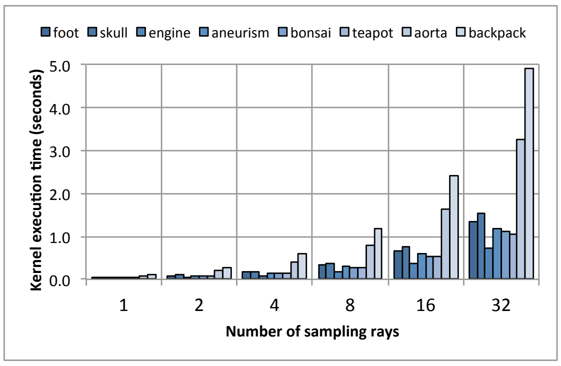
Lower resolution volume ray-casting can still provide a good trade-off between image quality and performance. Here, some experimental results for the *foot* and *backpack* datasets are presented, rendered in lower resolutions. The performance results are presented in Fig. 4.14, for one sampling ray. It is clear that the GPU-based implementation can easily achieve real-time visualization (30 fps) of volumetric datasets when the resulting image resolution is decreased, which means that fewer primary rays are used to sample the volume data. For a simple dataset (*foot*), interactive visualization (around 60 fps) can be achieved even for higher resolutions,

Table 4.5: High-resolution execution times for eight different datasets.

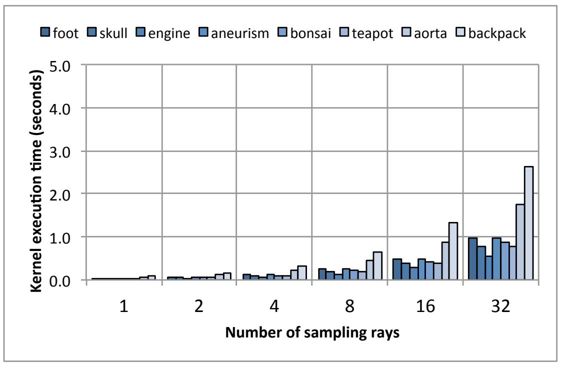
Data	Sampling rays, OpenMP Core i7					
	1	2	4	8	16	32
foot	2.54	4.68	9.02	18.30	34.58	69.75
skull	2.07	3.94	7.22	15.08	30.36	55.45
engine	2.14	4.12	8.63	16.81	33.42	66.45
aneurism	2.69	5.43	11.20	21.41	41.19	81.48
bonsai	2.23	4.19	7.41	14.65	30.16	55.78
teapot	2.58	4.57	8.72	16.84	38.36	73.00
aorta	6.19	12.22	24.08	47.94	95.76	192.46
backpack	6.36	12.51	24.97	49.45	99.22	198.32
Data	Sampling rays, 1× NVIDIA GTX 470					
	1	2	4	8	16	32
foot	0.03	0.08	0.17	0.34	0.67	1.35
skull	0.04	0.09	0.19	0.38	0.77	1.54
engine	0.02	0.04	0.09	0.18	0.37	0.74
aneurism	0.03	0.07	0.15	0.29	0.59	1.18
bonsai	0.03	0.06	0.14	0.27	0.53	1.11
teapot	0.03	0.06	0.13	0.26	0.53	1.06
aorta	0.08	0.19	0.39	0.80	1.62	3.26
backpack	0.12	0.29	0.58	1.20	2.43	4.91
Data	Sampling rays, 2× NVIDIA GTX 470					
	1	2	4	8	16	32
foot	0.03	0.06	0.12	0.24	0.48	0.96
skull	0.02	0.05	0.09	0.19	0.38	0.76
engine	0.02	0.03	0.07	0.14	0.28	0.56
aneurism	0.03	0.06	0.12	0.24	0.49	0.97
bonsai	0.03	0.05	0.11	0.22	0.44	0.87
teapot	0.02	0.05	0.09	0.19	0.38	0.77
aorta	0.06	0.12	0.23	0.45	0.88	1.76
backpack	0.08	0.17	0.33	0.66	1.32	2.62

as in Fig.4.14(a). On the other hand, the *backpack* dataset can achieve interactive visualization performance for very-low resolutions only, as depicted in Fig.4.14(b). Moreover, the OpenMP-based implementation cannot provide real-time or interactive rendering yet. Thus, optimizations are necessary in order to improve the algorithm performance in OpenMP, as shown in [82].

The MPSoC-based implementation results are shown in Fig. 4.15. Because of memory limitations of the FPGA, it could only render images of 640 × 480 pixels. Also, the *aorta* and *backpack* datasets could not fit in memory. In Fig. 4.15(a), one can observe that almost all the FPGA slices are being used (82%), as well as the available BlockRAMs (95%). Because of that, the FPGA could fit up to 4 microprocessors running in parallel. The high usage of BlockRAMs is due to the MPMC implementation of FIFOs for each input/output memory port, in order to

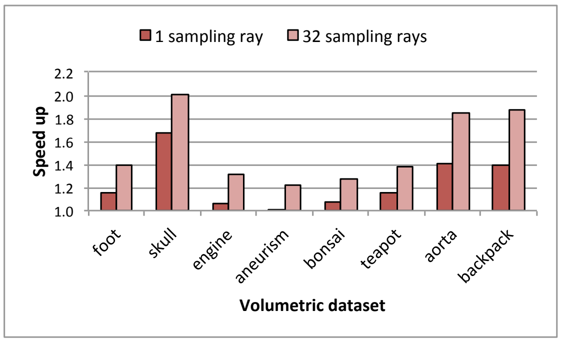


(a) Single-GPU, CUDA-based results.

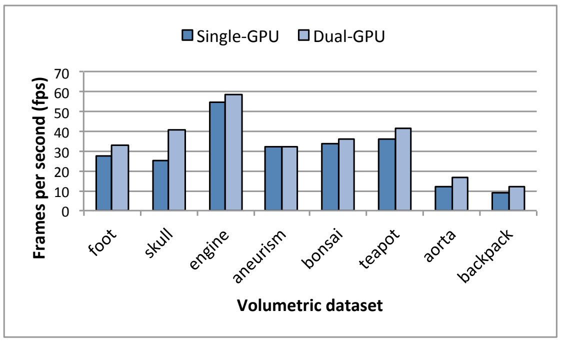


(b) Dual-GPU, CUDA-based results.

Figure 4.12: GPU performance results in CUDA.



(a) Single vs. Dual GPU speedup.



(b) GPU frame rate (1 sampling ray).

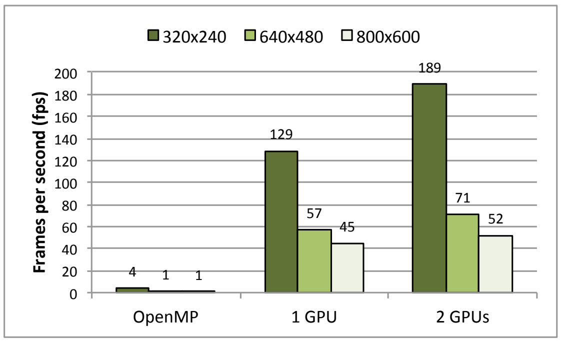
Figure 4.13: Acceleration rate using two GPUs and frames per second rate.

improve timing and performance [80].

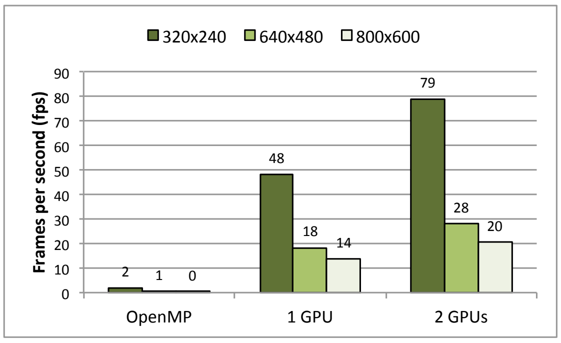
Performance and scalability results are shown in Fig. 4.15(b). For most datasets the parallel algorithm execution time improves as more MicroBlaze microprocessors (Processing Elements - PEs) are being used in parallel. The MPMC FIFOs for the fourth microprocessor are implemented using shift register lookup tables instead of BlockRAMs, which can contribute to create stalls in the datapath and, hence, worsen the overall performance of the microprocessor.

Finally, it is clear that interactive performance is not yet achieved. However, an Application-Specific Integrated Circuit (ASIC) implementation of such application-specific MPSoC design, instead of FPGA, could most probably run faster, with lower area and power consumption, as in [77].

Summing up, three un-optimized implementations of the volume ray-casting algorithm are discussed and compared. The GPU-based implementation is up to 90 times faster when a dual-GPU configuration is used, in comparison to the OpenMP-based implementation. One of the reasons for such speedup gain is because thousands of lightweight threads can be executed in parallel on GPU, while in the OpenMP-based implementation only 8 threads are executing in parallel. Fur-

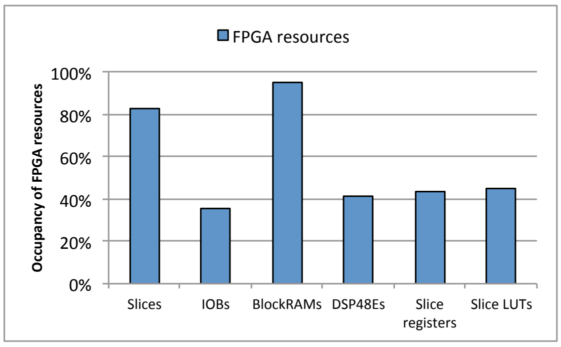


(a) *Foot* dataset low-resolution *fps.*

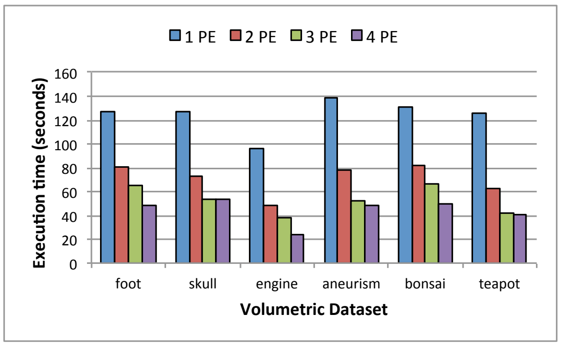


(b) *Backpack* dataset low-resolution *fps.*

Figure 4.14: Frames per second rendering rate for lower resolutions.



(a) FPGA area occupancy (4 PEs).



(b) Execution times for 640 × 480 res.

Figure 4.15: MPSoC synthesis and scalability, for up to 4 parallel microprocessors.

thermore, the overhead of changing between threads in GPU is much lower. The MPSoC-based implementation on a single Virtex-5 FPGA can execute up to four MicroBlaze microprocessors in parallel, running at 125MHz. As more microprocessors are used, the better is the performance achieved. However, interactive performance is not yet achieved, although in ASIC technology it could most probably run at higher frequencies, with more dedicated hardware.

This work demonstrated that the un-optimized GPU implementation of the volume ray-casting algorithm is able to deliver a high performance, between 60 and 90 times higher than that of the OpenMP-based implementation. For most datasets, high-resolution interactive visualization is achievable. Also, if the algorithm would make use of the texture and constant memories of the GPU, it would very likely achieve much higher frame rates, since the latency of these memories is much lower than the global memory latency. On the other hand, interactive performance may only be achieved in OpenMP unless several optimizations are applied to the algorithm. Furthermore, since the GPU implementation introduces more hardware overhead comparing to an MPSoC-based ASIC implementation, a MPSoC-based ASIC is expected to have lower area/power consumption.

4.2 ASIP-based Multi-Processor System-on-a-Chip

This Section presents a parallelization strategy of the ray-tracing algorithm to be implemented in two different reconfigurable Multi-Processor Systems-on-a-Chip [11, 13], as well as, hardware replication-aware instruction set extension based on the identification of equivalent computation patterns that are frequently executed in the ray-tracing application, as described in Chapter 2. Based on the information collected during the application profiling, which is performed using the LLVM compiler framework [42], candidates for instruction set extension are decided and implemented as specialized function units in the ASIP-based MPSoC to further speedup the ray-tracing execution time. Such extensions may substantially increase the circuit-area. To mitigate the impact on the area, the resource sharing optimization is used.

The ray-tracing algorithm is an example of a multimedia application highly-demanding in terms of throughput, circuit-area and energy consumption [83, 84]. First of all, the algorithm performance can be considerably improved by means of code and processor architecture parallelization [85]. Secondly, the algorithm may compute several ray-object intersection tests for every ray vector, which requires specific intersection computations/routines for each 3-D object of the whole 3-D scene [65]. Thus, in an ASIP design process, each intersection computation may be implemented as a specialized function unit in hardware, contributing to further speedup the overall execution. However, the more processing elements and specialized function units are added to the design, the higher becomes the overall system circuit-area and energy consumption. Nevertheless, the increase of the circuit-area and energy consumption can be limited through hardware re-use [10].

In contrast to traditional 3-D rendering algorithms [65], the ray tracing algorithm produces a higher fidelity image representation of a 3-D scene. For every primary ray (e.g. light vector), the ray tracing algorithm usually computes intersection tests against all the 3-D primitives (a.k.a. objects) of the scene, looking for the objects that are visible from a virtual camera's perspective. If an intersection is encountered, the object properties are used to determine wether the ray will be reflected, refracted or completely absorbed. For instance, if the ray is reflected or refracted, the algorithm is recursively executed to determine the objects that are visible from the previous intersection point perspective, which is why the algorithm can naturally produce mirror like effects in the final image. On the other hand, if the ray is absorbed, the processing ends and all the information that has been gathered until that point is merged to compose the color of the corresponding pixel of the viewplane. The program main entry is presented in Algorithm 6, in which the primary rays are being traced. The *trace* procedure in Algorithm 6 is responsible for determining the closest intersection point. Such procedure is recursively executed until a maximum reflection/refraction threshold is reached. Further details on ray tracing can be found in [65].

Algorithm 6 Ray Tracing primary rays

```
Require: 3-D primitives file
Ensure: rendered image
 1: scene = load3DScene(file);
 2: viewplane = setupViewplane(width,height);
 3: camera = setupCamera(viewplane,eye,view_direction);
 4: depth = 0;
 5: for $i \leftarrow 0$ to viewplane's width do
 6:     for $j \leftarrow 0$ to viewplane's height do
 7:         ray ← get(i,j,camera);
 8:         image[i][j] ← trace(scene,ray,depth);
```

4.2.1 Parallel ray-tracing in MPSoC

The parallel ray-tracing implementation is presented in Algorithm 7, where iterations of the external *for loop* are split across the microprocessors, as shown in line 5. In that way, groups of rays are assigned to different microprocessors, because every ray can be processed independently from the others.

Each microprocessor will produce different columns of the final rendered image. Every microprocessor knows which data to read and to write, according to its own identification number (CORE_ID$= 0, 1, 2, ..., n-1$) and also according to the total number of enabled microprocessors (N_CORES$= 1, 2, ..., n$), as shown in lines 5 and 9 of Algorithm 7. Observe that, at each inner-loop iteration, an image pixel is produced, as shown in line 8. There are no memory write conflicts, because the pixels produced by different microprocessors are always written at different memory addresses.

Algorithm 7 Parallel Ray Tracer in MPSoC

```
Require: 3-D primitives file
Ensure: rendered image
 1: scene = load3DScene(file);
 2: viewplane = setupViewplane(width,height);
 3: camera = setupCamera(viewplane,eye,view_direction);
 4: depth = 0;
 5: for $i \leftarrow$ CORE_ID to viewplane's width do
 6:     for $j \leftarrow 0$ to viewplane's height do
 7:         ray ← get(i,j,camera);
 8:         image[j + i * viewplane's height] = trace(scene, ray);
 9:     i ← i + N_CORES;
```

4.2.2 RISC-based MPSoC

The reconfigurable RISC-based MPSoC macro-architecture consists of several Xilinx MicroBlaze microprocessors running in parallel at 125MHz. They are connected to a shared DDR memory via a Xilinx Multi-Port Memory Controller (MPMC), which supports the connection of up to eight MicroBlaze microprocessors. Thus, the multi-port memory controller, together with the constraint on the available resources in the used FPGA, impose a limitation on the number of microprocessors that can actually be synthesized. The macro-architecture is depicted in Fig. 4.16.

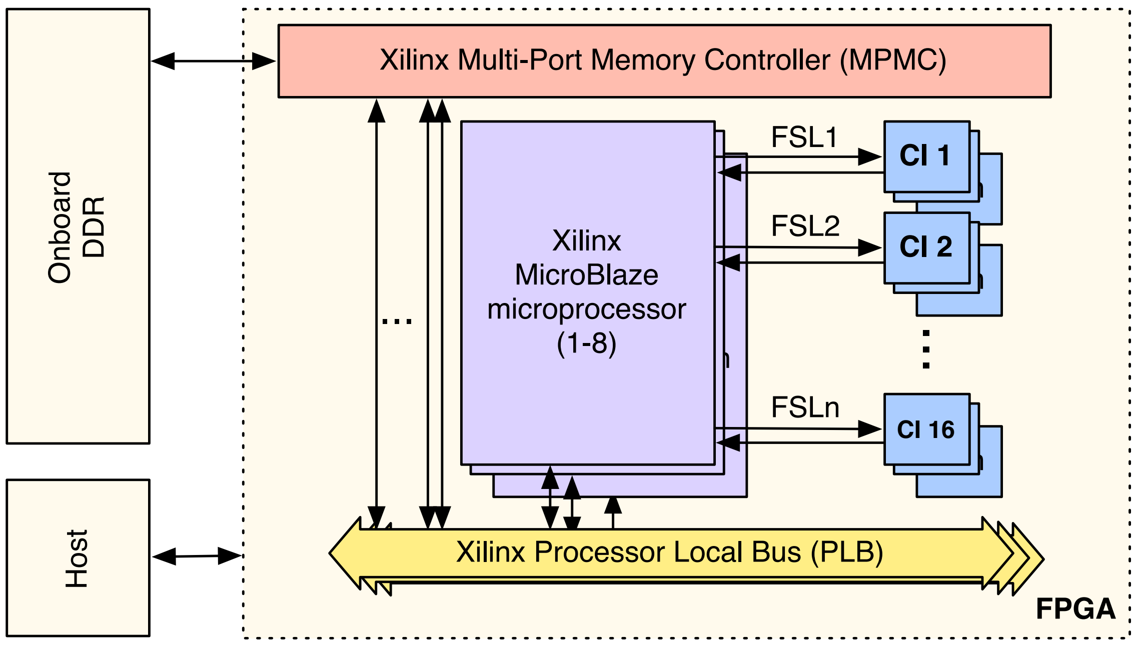

Onboard DDR
Host
Xilinx Multi-Port Memory Controller (MPMC)
Xilinx MicroBlaze microprocessor (1-8)
FSL1
FSL2
FSLn
CI 1
CI 2
CI 16
Xilinx Processor Local Bus (PLB)
FPGA

Figure 4.16: The reconfigurable MPSoC macro-architecture.

Each microprocessor's instruction set can be extended with up to 16 custom instructions, implemented as co-processors through the Xilinx Fast Simplex Link (FSL) bus. Thus, for the ray-tracing RISC-based MPSoC, each custom instruction works as a special floating-point co-processor.

4.2.2.1 Instruction set customization

Using the proposed LLVM-based instruction set customization tool, presented in Chapter 2, three of the most widely used floating-point custom instructions were added to each microprocessor instruction set, as shown in Fig. 4.17. The instruction extensions were selected from the most frequently executed operation patterns and accounting for the most common pattern occurrences found between the basic blocks during the profiling of the ray-tracing application. Finally, the custom instructions were manually mapped into the application source code, as shown in Listing 4.1.

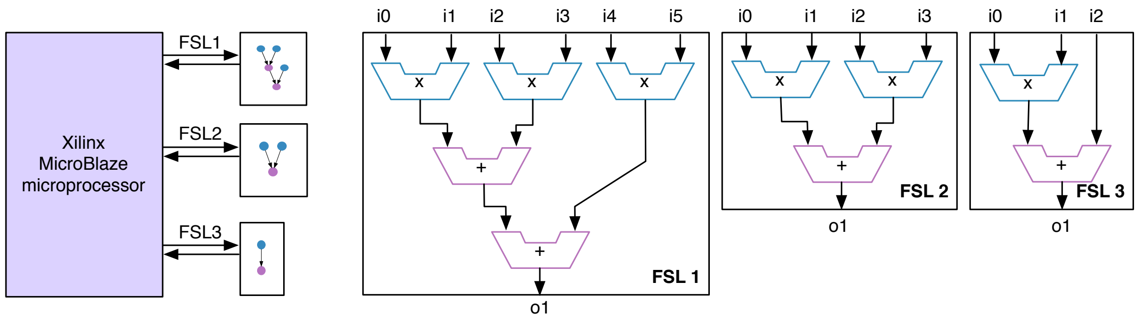


Figure 4.17: The Ray-Tracing instruction set extensions.

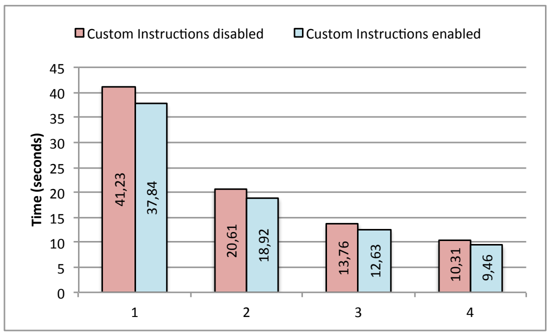
Listing 4.1: Multiply-add (ACC_MADD) custom instruction.

```
//connects to FSL2
#define ACC_MADD(X1,Y1,Z1,RES)\
{\
  asm volatile("
  put %1, rfsl2\n
  put %2, rfsl2\n
  put %3, rfsl2\n
  get %0, rfsl2":"=r"(RES):"r"(X1),"r"(Y1),"r"(Z1): );\
}
//ray-tracing "trace" function
color trace(camera cam, ray *r, int iter) {
  float lowest_dist;
  int index = -1;
  enum object obj = NONE;

  if (iter > MAX_DEPTH) { return black; }
  else {
    index = -1;
    obj = intersect3Dscene(&index, r, &lowest_dist);
    if (obj != NONE && lowest_dist > epsilon) {
      point intersection;
      point dir;
      dir.x = r->d.x - r->o.x;
      dir.y = r->d.y - r->o.y;
      dir.z = r->d.z - r->o.z;
      normalize(&dir);
      ACC_MADD(dir.x,lowest_dist,r->o.x,intersection.x);
      ACC_MADD(dir.y,lowest_dist,r->o.y,intersection.y);
      ACC_MADD(dir.z,lowest_dist,r->o.z,intersection.z);
      return shade(cam, &dir, obj, index, &intersection, iter);
    } else {
      return black;
    }
  }
  return black;
}
```

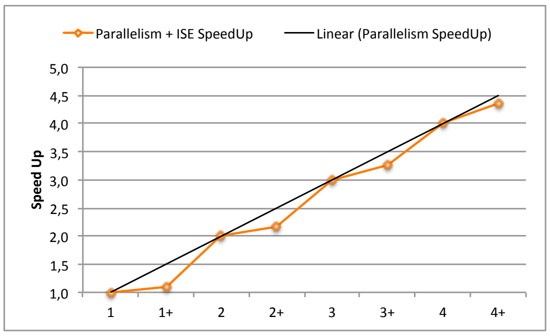
The RISC-based MPSoC macro-architecture, as described in Section 4.2.2, was synthesized using Xilinx EDK 14.4 for a Virtex-5 XC5VFX70T FPGA and the parallel algorithm implementation was compiled using MicroBlaze GCC compiler, without optimizations. Two implementations and experimental results are presented: the first one (Section 4.2.2.2) without ISE hardware sharing and the second (Section 4.2.2.3) with ISE hardware sharing.

4.2.2.2 Results of ISE without Hardware Sharing

The results are based on the ISE without hardware sharing exploration. In this ISE configuration, up to 4 MicroBlaze microprocessors were synthesized. The execution time results, shown in Fig. 4.18(a), are given in seconds and the speedup in comparison to a single microprocessor implementation is presented in Fig. 4.18(b). It is easy to observe that the speedup grows linearly with using more processing elements in parallel. Moreover, if the instruction set extensions are enabled, the speedup grows in the direction of the linear parallel speedup. Whenever they were enabled, the instruction set extensions provided altogether 8.2% speedup in any configuration of microprocessors. Four microprocessors with enabled instruction set extensions achieved 77% speedup in comparison to the standard single-processor solution.



(a) Execution time results, varying from 1 to 4 microprocessors.

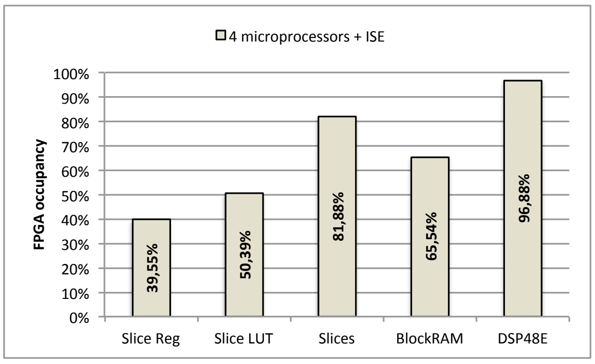


(b) Parallel ray-tracer speedup, varying from 1 to 4 microprocessors. A plus signal (+) indicates usage of ISE.

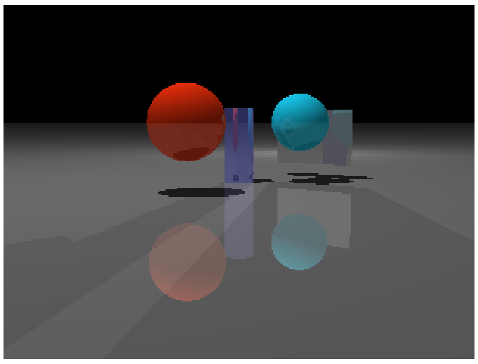
Figure 4.18: Parallel ray-tracer execution time results.

In the this design, almost all the FPGA slices are used (80%), as well as the available DSP48Es (81%), which are essential to lower the delay of the floating-point units in FPGA. Therefore, we could only fit in 4 microprocessors running in parallel with their instruction set extensions, as shown in Fig. 4.19(a). The resultant ray-traced image is presented in Fig. 4.19(b).

Furthermore, in order to better evaluate the impact/improvement due to the instruction set extensions, the complexity of the 3-D scene has been increased. Namely, we included 100 extra spheres in the 3-D scene. In this way, the number of required floating-point computations (during the intersection tests) has also increased. In this case, the instruction set extensions provided 10% speedup. Thus, the more data is fed into the custom instructions, the higher the speedup.



(a) FPGA resources usage for 4 microprocessors and peripherals.



(b) Ray-tracer output image (800 × 600 pixels).

Figure 4.19: RISC-based MPSoC FPGA area occupancy and the final output image.

4.2.2.3 Results of ISE with Hardware Sharing

Observe in Fig. 4.17 that each custom instruction presents a few function units in common. This is the same problem of maximal common (sub)-graph identification (common pattern identification), as discussed in Chapter 3. Therefore, we analyzed the proposed instruction set extension regarding its hardware sharing possibilities. The proposed instruction set extension tool was able to merge the common patterns and produce a compact function unit hardware that can still compute the custom instructions one at a time, as depicted in Fig. 4.20.

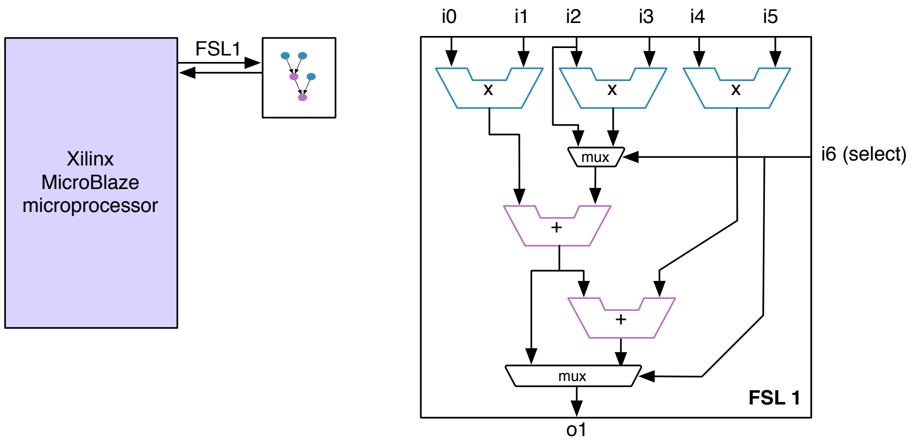
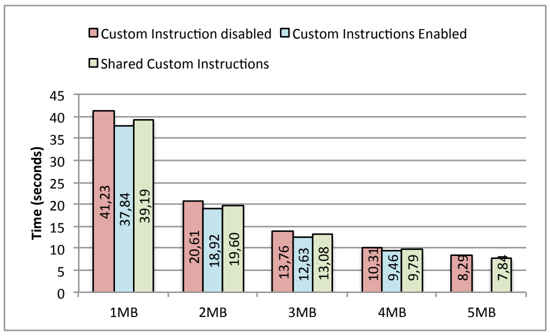
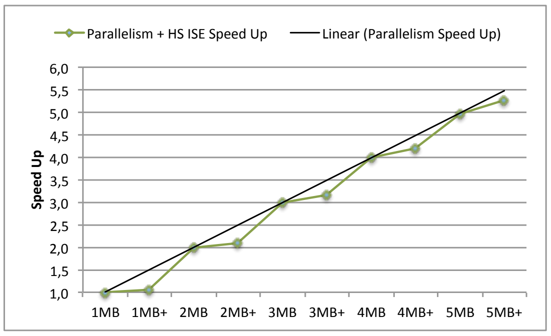


Figure 4.20: Instruction set extensions with Hardware Sharing.

The instruction set extensions with hardware sharing saved enough circuit-area to enable the inclusion of an additional microprocessor with custom instructions along to the other 4 microprocessors. Thus, using the Virtex 5 XC5VFX70T FPGA, up to 5 MicroBlaze microprocessors were included and synthesized. All the execution time results, shown in Fig. 4.21(a), are given in seconds and the speedup is in reference to a single microprocessor implementation is presented in Fig. 4.21(b).

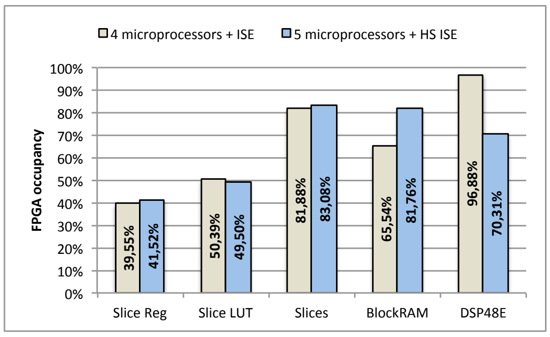


(a) Execution time results, varying from 1 to 5 microprocessors with hardware sharing.

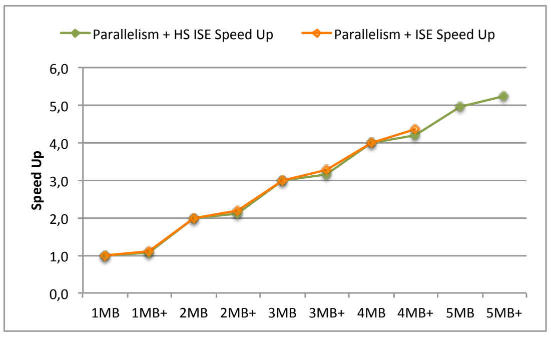


(b) Parallel ray-tracer speedup, varying from 1 to 5 microprocessors. A plus signal (+) indicates usage of ISE with HS.

Figure 4.21: RISC-MPSoC parallel ray-tracer execution time comparison, with hardware sharing.



(a) FPGA resources usage for 5 microprocessors with Hardware Sharing (HS).



(b) Speedup comparison, varying from 1 to 5 microprocessors. A plus signal (+) indicates usage of ISE.

Figure 4.22: RISC-MPSoC FPGA area occupancy and speedup results, with hardware sharing and without it.

The design with hardware sharing presented a better FPGA occupancy efficiency, as shown in Fig. 4.22(a). As expected, there is a very small loss of performance in the version of ISEs with hardware sharing, because the selection hardware requires an additional operation control signal to select which data-path should be followed at each time. Thus, altogether, the instruction set extensions with hardware sharing provided 5% speedup when enabled. The speedup is almost the same as that achieved by the ISEs with no hardware sharing, as shown in the comparison depicted in Fig. 4.22(b). Furthermore, the fifth microprocessor further improved the overall speedup to 81%, in comparison to the standard single-processor solution.

4.2.3 VLIW-based MPSoC

The multi-core VLIW-ASIP macro-architecture consists of several commercially available extensible VLIW-ASIPs (Cores) running in parallel, each with its own local memory. Each core is connected to the system's bus, enabling communication with the system's global memory. A host processor is also connected to the system's bus and, thus, can communicate with each core. The host processor is generally used to feed data into each core local memory and also into the system's global memory. Due to the highly-parallel nature of the ray-tracing and volume ray-casting algorithms, the cores do not need to communicate with each other, because each core can process a vector ray independently from the others. The macro-architecture is depicted in Fig. 4.23.

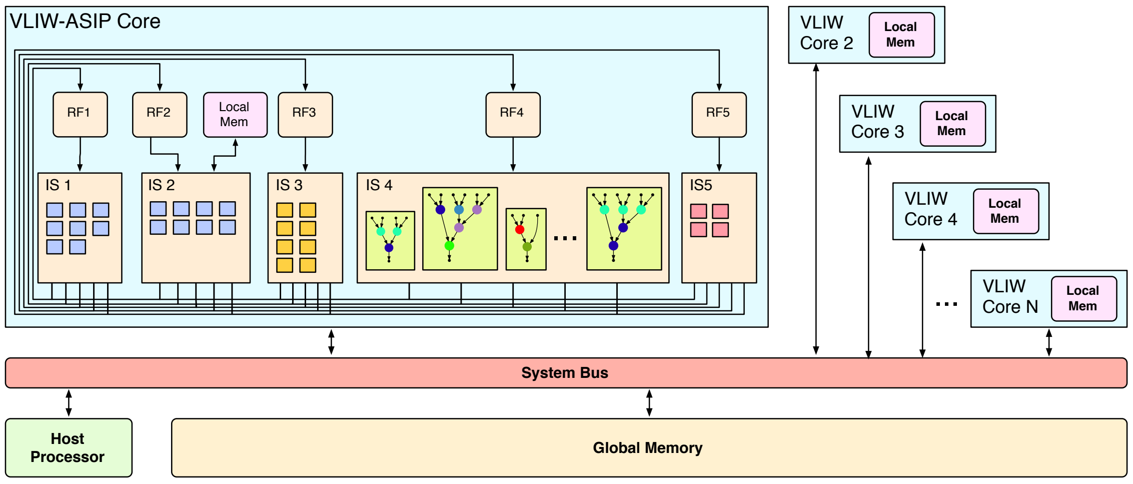


Figure 4.23: The Multi-core VLIW Macro-Architecture.

4.2.3.1 Instruction set customization

Inside each VLIW-ASIP there are issue-slots (IS), register files (RFs) and a local memory (64KB), as depicted in Fig. 4.24. The local memory is used to store frequently accessed data. Each issue slot can be designed to implement the datapath of a given set of operations, organized in function units (FU). For instance, in the proposed parallel architecture, the function units in issue-slots 1 and 2 implement the standard operations, i.e. load, store, addition, multiplication, etc. Issue-slot 3 implements the standard set of floating-point operations, whereas issue-slot 4 implements the special set of floating-point operations that were identified by the proposed automatic instruction set extension tool. Finally, issue-slot 5 implements the set of operations related to external issue-slot access, *e.g.*, global memory access. The instruction set extensions that were identified by the proposed automatic instruction set customization tool, during the application profiling, are presented in Appendix C.

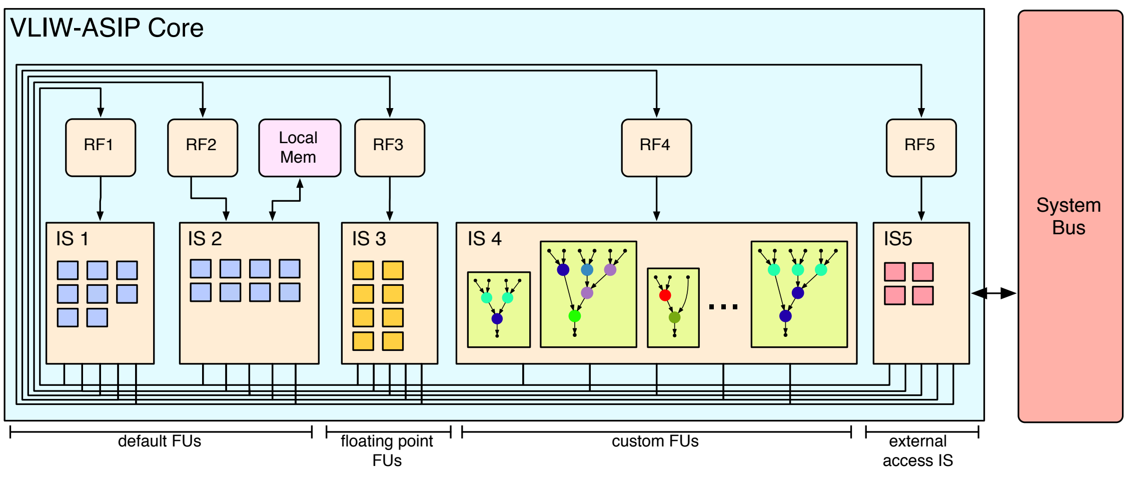


Figure 4.24: The MPSoC VLIW-ASIP Core.

4.2.3.2 Experimental Results

The performance results for the parallel ray-tracer application are presented in Fig. 4.25. As expected, the application benefits from parallelism exploitation: the speedup is almost linear as more VLIW-ASIPs are included in the parallel computation of rays, regardless of using instruction set extensions. When the instruction set extensions are enabled, the speedup goes beyond the expected linear speedup, pushing forward the VLIW-ASIP efficiency.

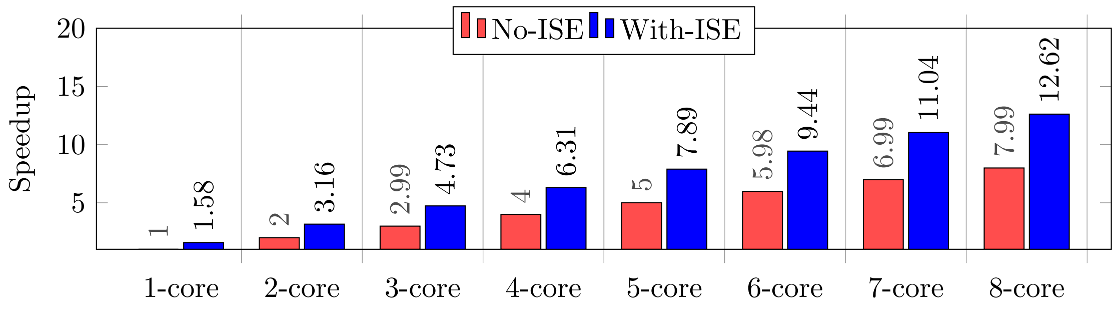


Figure 4.25: High-resolution ray-tracing speedup in respect to the number of VLIW-ASIP cores, with and without instruction set extensions.

Compared to most GPU's currently available in the market (with clocks speeds above 1GHz) the proposed parallel ray-processing architecture can achieve real-time performance at lower clock speeds. For instance, running at 200MHz, the architecture implementation with eight VLIW-Cores would be capable to compute one frame of the ray-tracer in 0.94 seconds with no specialized instructions and in 0.59 seconds when using specialized instructions, for a high-resolution scene of 1280 × 800 primary rays (resultant image pixels). Thus, lower resolutions can

	Ray-Tracing	
	No-ISE	ISE
1-core	1510692221	956876186
2-core	755362781	478449159
3-core	503941094	319195862
4-core	377747331	239268224
5-core	302184012	191404208
6-core	252517885	159944987
7-core	216069914	136861065
8-core	188940612	119678918

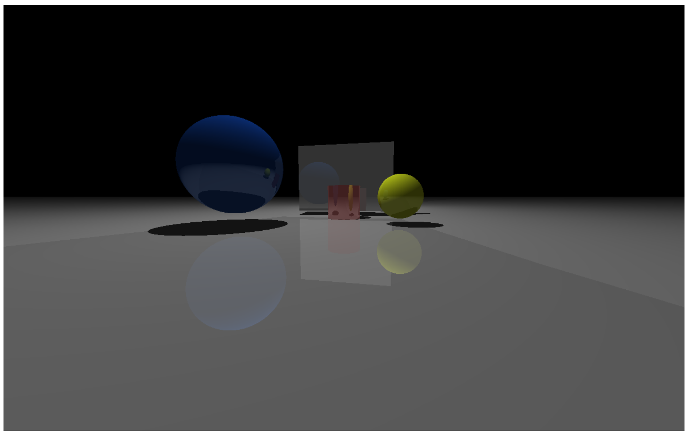
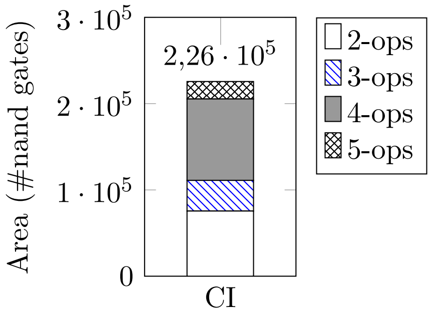
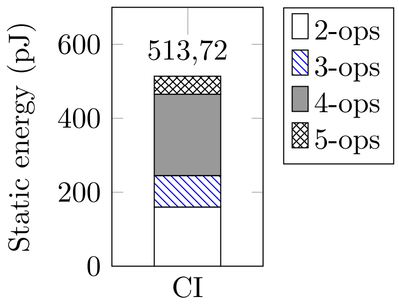


Figure 4.26: Produced image together with the cycle-accurate simulation results (number of cycles).



(a) Custom instructions circuit-area (in number of *nand gates*) for different granularities.



(b) Custom instructions static energy consumption (in pJ) for different granularities.

Figure 4.27: MPSoC VLIW-ASIP custom instructions circuit-area and energy consumption.

easily achieve interactive frame rates. The cycle-accurate simulation results and the resultant image are presented in Fig. 4.26.

The custom instructions circuit-area and energy consumption estimation for different granularities, *i.e.*, different sizes of custom instructions, are presented in Fig. 4.27. The circuit-area is given in number of *nand gates*, while the energy consumption is given in pico-joules (pJ). In total, the ray-tracing custom instructions require 225627 *nand gates*, with an static energy consumption of 513.72 pJ, approximately.

The proposed multi-core architecture based on a commercially available extensible VLIW-ASIP, augmented with an efficient instruction set customization for ray-tracing achieves as high as 12× speedup, in comparison to a single-core VLIW design, respectively. Also, the proposed instruction set extension method and tool was able to identify and select the most promising operation patterns (sets of basic operations) throughout performing an application data-flow graph analysis. With only a few additional application-specific instructions, the ray-tracer execution time lowered in approximately 36%.

Chapter 5

Conclusions and ideas for future work

The design of an efficient ASIP is a difficult task, which is very dependent on an adequate application analysis in order to identify the main aspects and characteristics of the application or application domain for which the ASIP is being designed. This thesis proposed and discussed an efficient automatic instruction set extension tool, that is able to identify the most promising operation patterns (sets of basic operations) performing an application data-flow graph analysis at the basic block level. It focused on the automatic identification of custom instructions at the LLVM-IR level, using an exact and efficient subgraph isomorphism algorithm based on compatibility graphs and clique enumeration. The proposal is novel considering the graph connectivity and (re)-associativity detection aspects. An experimental analysis of the custom instructions identified by the proposed automated tool-set was presented for different custom instruction granularities, with cycle-accurate speedup results based on a commercially available extensible VLIW-ASIP for several important applications, together with area and energy consumption estimation results for a 65nm TSMC technology, running at 200MHz.

The identified application-specific operation patterns promoted substantial speedup for a set of well-known benchmark applications. When the custom instructions are enabled using one augmented issue-slot, the speedup is substantial and goes up to 25% for the RT application, using the 14 custom instructions that the tool automatically identified for it. Other applications, such as the CRC, achieved a speedup of 14% using only one custom instructions automatically identified for it. Using different configurations of augmented issue-slots to increase ILP, good results can be produced that involves substantial tradeoffs between area, energy and performance. For some applications, the results suggest that increasing the number of augmented issue-slots does not improve the application performance, while for others, using only a pair of such issue-slots is more than enough to achieve significant speedups. When the custom instructions are enabled using two augmented issue-slots, the speedup increases to 50% for the RT application, but does not change for the CRC and EDGE applications.

Replicating issue-slots to exploit ILP requires more circuit-area and imposes higher energy consumption. Nonetheless, a trade-off between performance, circuit-area and energy can always be found. It can be noted that the cost of including the custom instructions into the ASIP datapath regarding the AES application in terms of circuit-area and energy consumption is not prohibitively high when compared to that of the RT and VRC applications. Hence, replicating augmented issue-slots for the AES application in order to achieve higher speedups may seem to be more feasible in terms of circuit-area and energy consumption than in the case of the RT and VRC applications.

Regarding the granularity of the custom instructions, it could be noted that the fine-grain custom instructions contribute the most for the speedup of the applications, due to the fact that they got mapped more often onto the application data-flow graph. For instance, for some applications, the fine-grain custom instructions were mapped tens to hundreds of times more often than the coarse-grain ones. In some cases, the coarse-grain custom instructions had to mapped manually, because the VLIW retargetable compiler could not see the opportunities for mapping one or another coarse-grain custom instruction.

This thesis also discussed the problem of hardware sharing in ASIPs and accelerator synthesis, with focus on datapath merging techniques. Despite various hardware sharing optimizations that can be performed at earlier synthesis stages there is usually a substantial hardware sharing potential in the datapath synthesis stage that is not fully exploited. Besides, it showed that commercially available synthesis tools can only perform a limited set of hardware sharing optimizations, as well as circuit speed and energy consumption optimizations. This is an inherent limitation of the synthesis tools, which are based on several intractable problems for which there is no polynomial time solution. Therefore, several heuristics, such as the datapath merging technique, can be employed to find adequate optimization solutions in a reasonable time.

Finally, efficient ASIP-based multiprocessor systems-on-a-chip with complex instructions speedup have been designed for a class of highly demanding multimedia applications, such as ray-tracing. Such application can benefit from parallel processing and specialized custom instructions to speedup its execution. The RISC-based MPSoC is capable of achieving up to 5$\times$ speedup, while the VLIW-based MPSoC achieves up to 12$\times$ speedup, both with complex instructions enabled and using up to 8 parallel ASIPs. It was noted that the complex instructions contributed to lower the ray-tracing execution time by 5% to 36% for the RISC-based and the

VLIW-based MPSoCs, respectively. Also, the datapath merging hardware sharing optimization applied to the complex instructions of the RISC-based MPSoC saved enough area to enable the inclusion of an additional ASIP with complex instructions, which demonstrates the importance of hardware sharing in the context of instruction set extension and ASIPs design.

In the future, the custom instruction identification tool can be extended to include support for distributivity detection among the identified patterns. Furthermore, the subset of LLVM-IR basic operations that is currently accepted by the tool can also be extended, possibility including, for instance, *branch* operations and *load/store* operations. Unfortunately, the latter extension would require substantial modifications of the VLIW-ASIP architecture and compiler, that currently does not support *branch* and *load/store* custom function units.

In the hardware sharing context, exploitation of datapath merging can influence the area of ASIP and accelerator designs to a very high degree and may result in substantial area/power/delay tradeoffs. Thus, adequate hardware sharing techniques involving datapath merging are of high importance for the multi-objective ASIP and accelerator optimization and adequate tradeoff exploitation. For instance, in the compatibility graph datapath merging technique, an exploration method should be devised to enumerate and analyze all the cliques for each compatibility graph that is produced by each pair of directed acyclic graphs, enabling the selection of the best merging solution.

Bibliography

[1] JÓŹWIAK, L., NEDJAH, N., FIGUEROA, M. "Modern development methods and tools for embedded reconfigurable systems: A survey", *Integr. VLSI J.*, v. 43, pp. 1–33, January 2010. ISSN: 0167-9260.

[2] JÓŹWIAK, L., NEDJAH, N. "Modern Architectures for Embedded Reconfigurable Systems - a Survey", *Journal of Circuits, Systems, and Computers*, v. 18, n. 2, pp. 209–254, 2009.

[3] HANIKA, J., KELLER, A. "Towards Hardware Ray Tracing using Fixed Point Arithmetic". In: *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on*, pp. 119 –128, sept. 2007.

[4] CHU, P. P. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability.* Newark, NJ, Wiley-IEEE Press, 2006.

[5] RAYMOND, J. W., WILLETT, P. "Maximum common subgraph isomorphism algorithms for the matching of chemical structures", *Journal of Computer-Aided Molecular Design*, v. 16, pp. 2002, 2002.

[6] CHENG, J., ZHU, L., KE, Y., et al. "Fast algorithms for maximal clique enumeration with limited memory". In: *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '12, pp. 1240–1248, New York, NY, USA, 2012. ACM.

[7] PATTABIRAMAN, B., PATWARY, M. M. A., GEBREMEDHIN, A. H., et al. "Fast Algorithms for the Maximum Clique Problem on Massive Sparse Graphs", *CoRR*, v. abs/1209.5818, 2012.

[8] ARNOLD, M., CORPORAAL, H. "Designing domain-specific processors". In: *Proceedings of the ninth international symposium on Hardware/software codesign*, CODES '01, pp. 61–66, New York, NY, USA, 2001. ACM.

[9] JÓŹWIAK, L., GAWEOWSKI, D., SLUSARCZYK, A., et al. "Static Power Reduction in Nano CMOS Circuits Through an Adequate Circuit Synthesis".

In: *Mixed Design of Integrated Circuits and Systems, 2007. MIXDES '07. 14th International Conference on*, pp. 172 –177, june 2007.

[10] NERY, A., JÓŹWIAK, L., LINDWER, M., et al. "Hardware Reuse in Modern Application-Specific Processors and Accelerators". In: *Digital System Design (DSD), 2011 14th Euromicro Conference on*, pp. 140 –147, 31 2011-sept. 2 2011.

[11] NERY, A., NEDJAH, N., FRANA, F., et al. "A Reconfigurable Ray-Tracing Multi-Processor SoC with Hardware Replication-Aware Instruction Set Extension". In: Ko?odziej, J., Di Martino, B., Talia, D., et al. (Eds.), *Algorithms and Architectures for Parallel Processing*, v. 8285, *Lecture Notes in Computer Science*, Springer International Publishing, pp. 346–356, 2013.

[12] NERY, A., NEDJAH, N., FRANCA, F., et al. "Automatic complex instruction identification for efficient application mapping onto ASIPs". In: *Circuits and Systems (LASCAS), 2014 IEEE 5th Latin American Symposium on*, pp. 1–4, Feb 2014.

[13] NERY, A., NEDJAH, N., FRANCA, F., et al. "A Framework for Automatic Custom Instruction Identification on Multi-Issue ASIPs". In: *Industrial Informatics (INDIN), 12th IEEE International Conference on*, pp. 428–433, June 2014.

[14] GALUZZI, C., BERTELS, K. "The Instruction-Set Extension Problem: A Survey", *ACM Trans. Reconfigurable Technol. Syst.*, v. 4, n. 2, pp. 18:1–18:28, maio 2011. ISSN: 1936-7406.

[15] POZZI, L., ATASU, K., IENNE, P. "Exact and approximate algorithms for the extension of embedded processor instruction sets", *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, v. 25, n. 7, pp. 1209–1229, July 2006.

[16] ARORA, N., CHANDRAMOHAN, K., POTHINENI, N., et al. "Instruction Selection in ASIP Synthesis Using Functional Matching". In: *VLSI Design, 2010. VLSID '10. 23rd International Conference on*, pp. 146–151, Jan 2010.

[17] SHUAI LU, Y., SHEN, L., BO HUANG, L., et al. "Customizing computation accelerators for extensible multi-issue processors with effective optimization techniques". In: *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pp. 197–200, June 2008.

[18] BRISK, P., KAPLAN, A., SARRAFZADEH, M. "Area-efficient instruction set synthesis for reconfigurable system-on-chip designs". In: *Design Automation Conference, 2004. Proceedings. 41st*, pp. 395 –400, july 2004.

[19] ZULUAGA, M., TOPHAM, N. "Resource Sharing in Custom Instruction Set Extensions". In: *Application Specific Processors, 2008. SASP 2008. Symposium on*, pp. 7–13, June 2008.

[20] ZULUAGA, M., TOPHAM, N. "Design-space exploration of resource-sharing solutions for custom instruction set extensions", *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, v. 28, pp. 1788–1801, December 2009. ISSN: 0278-0070.

[21] ZULUAGA, M., TOPHAM, N. "Exploring the unified design-space of custom-instruction selection and resource sharing". In: *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pp. 282 –291, july 2010.

[22] ATASU, K., POZZI, L., IENNE, P. "Automatic Application-Specific Instruction-Set Extensions Under Microarchitectural Constraints", *International Journal of Parallel Programming*, v. 31, n. 6, pp. 411–428, 2003. ISSN: 0885-7458.

[23] BONZINI, P., POZZI, L. "A Retargetable Framework for Automated Discovery of Custom Instructions". In: *Application-specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, pp. 334–341, July 2007.

[24] ATASU, K., MENCER, O., LUK, W., et al. "Fast custom instruction identification by convex subgraph enumeration". In: *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*, pp. 1–6, July 2008.

[25] XIAO, C., CASSEAU, E. "Efficient maximal convex custom instruction enumeration for extensible processors". In: *Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on*, pp. 1–7, Nov 2011.

[26] LI, T., SUN, Z., JIGANG, W., et al. "Fast enumeration of maximal valid subgraphs for custom-instruction identification". In: *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, pp. 29–36. ACM, 2009.

[27] BONZINI, P., POZZI, L. “Polynomial-time Subgraph Enumeration for Automated Instruction Set Extension”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE ’07, pp. 1331–1336, San Jose, CA, USA, 2007. EDA Consortium.

[28] MOREANO, N., BORIN, E., DE SOUZA, C., et al. “Efficient datapath merging for partially reconfigurable architectures”, *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, v. 24, n. 7, pp. 969 – 980, july 2005.

[29] MOREANO, N. *Algoritmos para Alocação de Recursos em Arquiteturas Reconfiguráveis.* Tese de Doutorado, Instituto de Computação – Unicamp, São Paulo, Brazil, November 2005.

[30] BONZINI, P., POZZI, L. “Recurrence-Aware Instruction Set Selection for Extensible Embedded Processors”, *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, v. 16, n. 10, pp. 1259–1267, Oct 2008.

[31] TAN, H., SUN, Y. “Automatic identification of customized instruction based on multiple attribute decision-making for multi-issue architectures”, *Tsinghua Science and Technology*, v. 16, n. 3, pp. 278–284, June 2011.

[32] WU, I.-W., CHEN, Z.-Y., SHANN, J.-J., et al. “Instruction Set Extension Exploration in Multiple-issue Architecture”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE ’08, pp. 764–769, New York, NY, USA, 2008. ACM.

[33] WU, I.-W., SHANN, J. J.-J., HSU, W.-C., et al. “Extended Instruction Exploration for Multiple-Issue Architectures”, *ACM Trans. Embed. Comput. Syst.*, v. 13, n. 4, pp. 92:1–92:28, mar. 2014. ISSN: 1539-9087.

[34] JAIN, D., KUMAR, A., POZZI, L., et al. “Automatically Customising VLIW Architectures with Coarse Grained Application-Specific Functional Units”. In: Schepers, H. (Ed.), *Software and Compilers for Embedded Systems*, v. 3199, *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 17–32, 2004.

[35] NERY, A. S., JÓZWIAK, L., LINDWER, M., et al. “Hardware Reuse in Modern Application-specific Processors and Accelerators”, *Microprocess. Microsyst.*, v. 37, n. 6-7, pp. 684–692, ago. 2013. ISSN: 0141-9331.

[36] LIN, H., FEI, Y. “Resource sharing of pipelined custom hardware extension for energy-efficient application-specific instruction set processor design”.

In: *Proceedings of the 2009 IEEE international conference on Computer design*, ICCD'09, pp. 158–165, Piscataway, NJ, USA, 2009. IEEE Press.

[37] MOREANO, N., ARAUJO, G., HUANG, Z., et al. "Datapath merging and interconnection sharing for reconfigurable architectures". In: *System Synthesis, 2002. 15th International Symposium on*, pp. 38 –43, oct. 2002.

[38] ZULUAGA, M., KLUTER, T., BRISK, P., et al. "Introducing control-flow inclusion to support pipelining in custom instruction set extensions". In: *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, pp. 114 –121, july 2009.

[39] BERGROTH, L., HAKONEN, H., RAITA, T. "A survey of longest common subsequence algorithms". In: *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pp. 39 –48, 2000.

[40] GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., et al. "MiBench: A Free, Commercially Representative Embedded Benchmark Suite". In: *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pp. 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[41] JÓŹWIAK, L., LINDWER, M., CORVINO, R., et al. "ASAM: Automatic Architecture Synthesis and Application Mapping". In: *DSD 2012 - 15th Euromicro Conference on Digital System Design*, pp. 216–225, Cesme, Izmir, Turkey, September 2012.

[42] LATTNER, C., ADVE, V. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pp. 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[43] CSARDI, G., NEPUSZ, T. "The igraph software package for complex network research", *InterJournal*, v. Complex Systems, pp. 1695, 2006.

[44] BONDY, J. A., MURTY, U. *Graph Theory With Applications.* Oxford, UK, UK, Elsevier Science Ltd., 1976.

[45] KOCH, I. "Enumerating all connected maximal common subgraphs in two graphs", *Theoretical Computer Science*, v. 250, n. 1, pp. 1–30, 2001.

[46] BRON, C., KERBOSCH, J. "Algorithm 457: finding all cliques of an undirected graph", *Commun. ACM*, v. 16, n. 9, pp. 575–577, set. 1973. ISSN: 0001-0782.

[47] VILLA, O., CHAVARRA-MIR, D., GURUMOORTHI, V., et al. "Effects of Floating-Point non-Associativity on Numerical Computations on Massively Multithreaded Systems". In: *Cray User Group Meeting*, CUG 2009, Atlanta, Georgia, May 2009.

[48] KILTS, S. *Advanced FPGA Design: Architecture, Implementation, and Optimization*. Wiley-IEEE Press, 2007.

[49] XILINX. *Xilinx Synthesis Technology: User Guide*. http://www.xilinx.com/itp/xilinx5/pdf/docs/xst/xst.pdf, 2008. Disponível em: <`http://www.xilinx.com/itp/xilinx5/pdf/docs/xst/xst.pdf`>. January, 2009.

[50] HADJIS, S., CANIS, A., ANDERSON, J. H., et al. "Impact of FPGA architecture on resource sharing in high-level synthesis". In: *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, FPGA '12, pp. 111–114, New York, NY, USA, 2012. ACM.

[51] CANIS, A., CHOI, J., ALDHAM, M., et al. "LegUp: high-level synthesis for FPGA-based processor/accelerator systems". In: *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, pp. 33–36, New York, NY, USA, 2011. ACM.

[52] COMPTON, K., HAUCK, S. "Automatic Design of Area-Efficient Configurable ASIC Cores", *IEEE Trans. Comput.*, v. 56, n. 5, pp. 662–672, maio 2007. ISSN: 0018-9340.

[53] PEÑALBA, O., MENDÍAS, J. M., HERMIDA, R. "A global approach to improve conditional hardware reuse in high-level synthesis", *Journal of Systems Architecture*, v. 47, pp. 959–975, June 2002. ISSN: 1383-7621.

[54] ZULUAGA, M., TOPHAM, N. "Exploring the unified design-space of custom-instruction selection and resource sharing". In: *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pp. 282–291, July 2010.

[55] PEYMANDOUST, A., POZZI, L., IENNE, P., et al. "Automatic instruction set extension and utilization for embedded processors". In: *Application-Specific Systems, Architectures, and Processors, 2003. Proceedings. IEEE International Conference on*, pp. 108 – 118, june 2003.

[56] KOREN, I. *Computer arithmetic algorithms.* 2ª ed. Upper Saddle River, NJ, USA, Prentice-Hall, Inc., 2001.

[57] PARHAMI, B. *Computer arithmetic: algorithms and hardware designs.* Oxford, UK, Oxford University Press, 2009.

[58] AL-ERYANI, J. *Floating Point Unit Core.* http://opencores.org/project,fpu100, 2006. Disponível em: <`http://opencores.org/project,fpu100`>. Last access in: August 2011.

[59] DICK, R., RHODES, D., WOLF, W. "TGFF: task graphs for free". In: *Hardware/Software Codesign, 1998. (CODES/CASHE '98) Proceedings of the Sixth International Workshop on*, pp. 97 –101, mar 1998.

[60] NERY, A. S., NEDJAH, N., FRANÇA, F. M. G., et al. "Parallel Processing of Intersections for Ray-tracing in Application-specific Processors and GPGPUs", *Microprocess. Microsyst.*, v. 37, n. 6-7, pp. 739–749, ago. 2013. ISSN: 0141-9331.

[61] NERY, A. S., NEDJAH, N., FRANA, F. M., et al. "Interactive Volume Rendering Based on Ray-Casting for Multi-core Architectures". In: Dayd, M., Marques, O., Nakajima, K. (Eds.), *High Performance Computing for Computational Science - VECPAR 2012*, v. 7851, *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 177–186, 2013.

[62] NERY, A. S., NEDJAH, N., FRANÇA, F. M. G. "A Parallel Architecture for Ray-Tracing". In: *IEEE Latin-American Symposium on Circuits and Systems - LASCAS'10*, pp. 96–99, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[63] NERY, A. S., NEDJAH, N., FRANÇA, F. M. G. "A massively parallel hardware architecture for ray-tracing", *International Journal of High Performance Systems Architecture 2009 - Vol. 2, No.1 pp. 26 - 34*, pp. 26–34, 2009.

[64] FUJIMOTO, A., TANAKA, T., IWATA, K. "ARTS: accelerated ray-tracing system". pp. 148–159, New York, NY, USA, Computer Science Press, Inc., 1988.

[65] AKENINE-MÖLLER, T., HAINES, E., HOFFMAN, N. *Real-Time Rendering.* 3rd ed. Natick, MA, USA, A. K. Peters, Ltd., 2008.

[66] XILINX. "MicroBlaze Processor Reference Guide". `http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf`, 2008. Last acess: november, 2009.

[67] XILINX. "Fast Simplex Link v2.11b". `http://www.xilinx.com/support/documentation/ip_documentation/fsl_v20.pdf`, 2009. Last access: november, 2009.

[68] XILINX. "XPS UARTLite". `http://www.xilinx.com/support/documentation/ip_documentation/xps_uartlite.pdf`, 2009. Last access: november, 2009.

[69] KIRK, D. B., HWU, W.-M. W. *Programming Massively Parallel Processors: A Hands-on Approach.* San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2010.

[70] XILINX. "Floating-Point Operator v5.0". 2008. Disponível em: <`http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf`>. Last access: january, 2009.

[71] LABORATORY, S. C. G. "The Stanford 3D Scanning Repository". `http://www-graphics.stanford.edu/data/3Dscanrep/`, 2009. Last access: february, 2009.

[72] LACROUTE, P., LEVOY, M. "Fast volume rendering using a shear-warp factorization of the viewing transformation". In: *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, pp. 451–458, New York, NY, USA, 1994. ACM.

[73] BHANIRAMKA, P., DEMANGE, Y. "OpenGL volumizer: a toolkit for high quality volume rendering of large data sets". In: *Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, VVS '02, pp. 45–54, Piscataway, NJ, USA, 2002. IEEE Press.

[74] LORENSEN, W., CLINE, H. "Marching cubes: A high resolution 3D surface construction algorithm", *SIGGRAPH Comput. Graph.*, v. 21, pp. 163–169, August 1987. ISSN: 0097-8930.

[75] LEVOY, M. "Efficient ray tracing of volume data", *ACM Trans. Graph.*, v. 9, pp. 245–261, July 1990. ISSN: 0730-0301.

[76] WALD, I., FRIEDRICH, H., MARMITT, G., et al. "Faster Isosurface Ray Tracing Using Implicit KD-Trees", *IEEE Transactions on Visualization and Computer Graphics*, v. 11, n. 5, pp. 562–572, 2005.

[77] PFISTER, H., HARDENBERGH, J., KNITTEL, J., et al. "The VolumePro Real-time Ray-casting System". In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pp. 251–260, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co. ISBN: 0-201-48560-5.

[78] MENSMANN, J., ROPINSKI, T., HINRICHS, K. H. "An Advanced Volume Raycasting Technique using GPU Stream Processing". In: *GRAPP: International Conference on Computer Graphics Theory and Applications*, pp. 190–198, Angers, 2010. INSTICC Press.

[79] COX, G., SILVA, C., CUPERTINO, L., et al. "Exploring Parallelism in Volume Ray Casting: Understanding the Programming Issues of Multithreaded Accelerators". In: *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '12, pp. 64–73, New York, NY, USA, 2012. ACM. ISBN: 978-1-4503-1211-0.

[80] XILINX. "LogiCORE IP Multi-Port Memory Controller (MPMC) v6.03.a". `http://www.xilinx.com/support/documentation/ip_documentation/mpmc.pdf`. Last access: May, 2012.

[81] BARTZ. "Volvis – Volume Library". `http://www.volvis.org/`, 2005. Last access: May, 2012.

[82] LEE, V. W., KIM, C., CHHUGANI, J., et al. "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU", *SIGARCH Comput. Archit. News*, v. 38, n. 3, pp. 451–460, jun. 2010.

[83] KOPTA, D., SHKURKO, K., SPJUT, J., et al. "An energy and bandwidth efficient ray tracing architecture". In: *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pp. 121–128, New York, NY, USA, 2013. ACM.

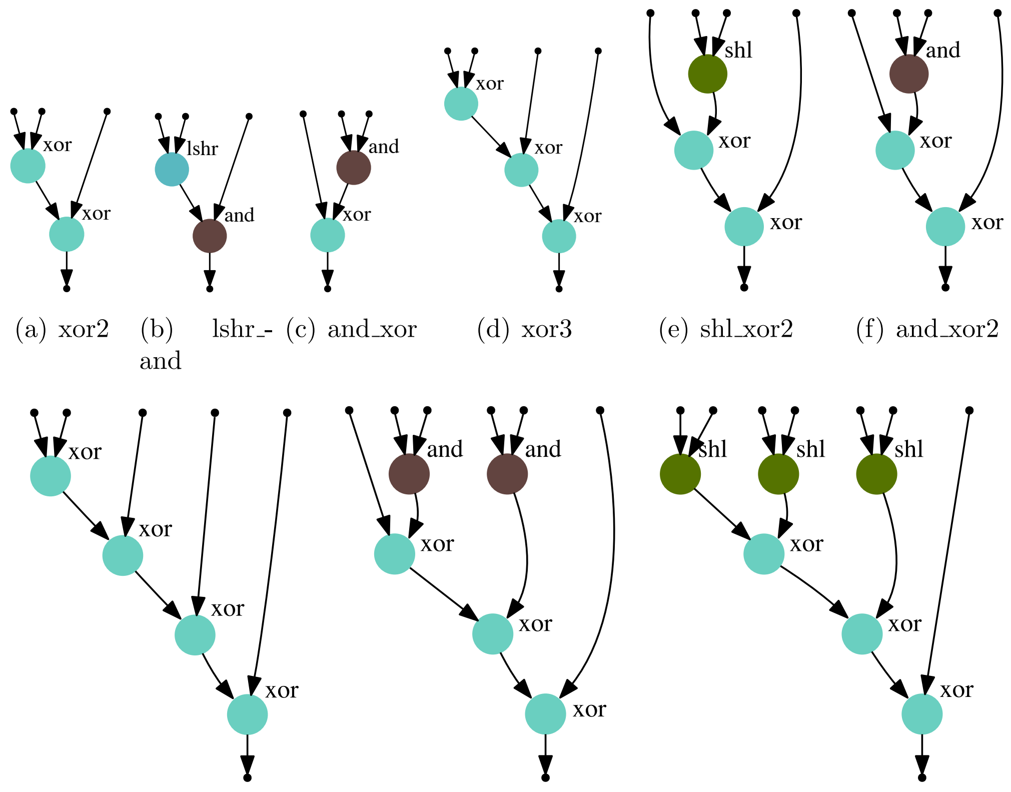
[84] LEE, W.-J., SHIN, Y., LEE, J., et al. "SGRT: a mobile GPU architecture for real-time ray tracing". In: *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pp. 109–119, New York, NY, USA, 2013. ACM.

[85] GRIBBLE, C., FISHER, J., EBY, D., et al. "Ray tracing visualization toolkit". In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D*

Graphics and Games, I3D '12, pp. 71–78, New York, NY, USA, 2012. ACM.

Appendix A

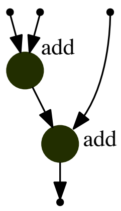
Library of custom instructions



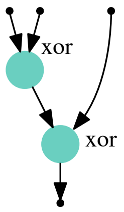
(a) xor2 (b) lshr_and (c) and_xor (d) xor3 (e) shl_xor2 (f) and_xor2

(g) xor4 (h) c_sp1 (i) b_sp2

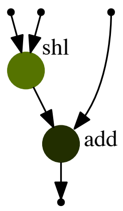
Figure A.1: AES custom instructions.



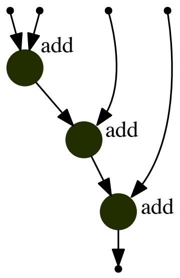
(a) add_add



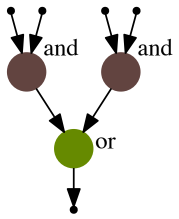
(b) xor_xor



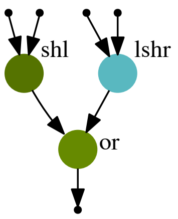
(c) shl_add



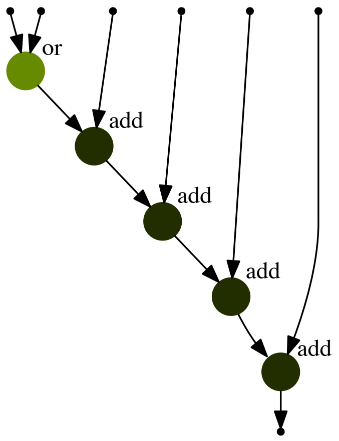
(d) add_add_add



(e) v_and_or_and

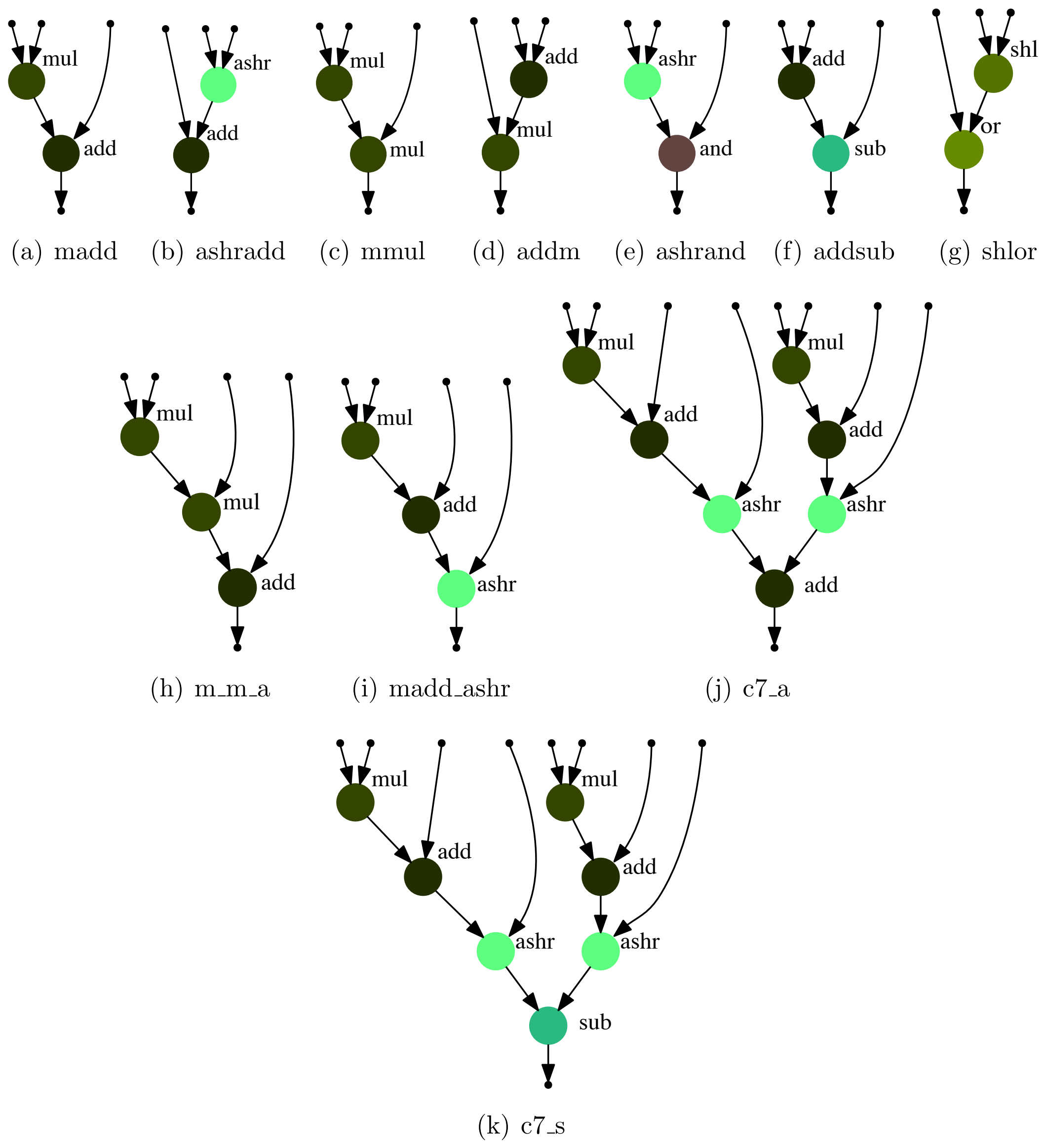


(f) v_shl_or_lshr



(g) or_l_add

Figure A.2: SHA custom instructions.

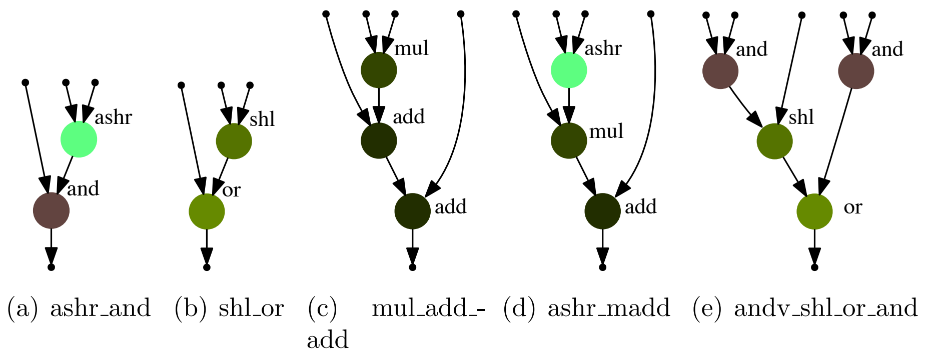


(a) madd (b) ashradd (c) mmul (d) addm (e) ashrand (f) addsub (g) shlor

(h) m_m_a (i) madd_ashr (j) c7_a

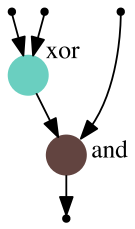
(k) c7_s

Figure A.3: JPEG custom instructions.



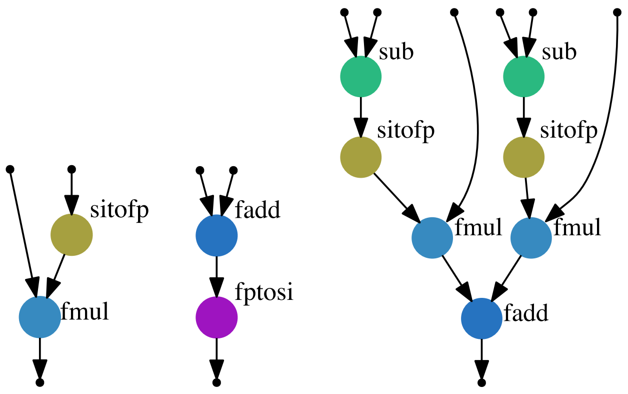
(a) ashr_and (b) shl_or (c) mul_add_-add (d) ashr_madd (e) andv_shl_or_and

Figure A.4: MJPEG custom instructions.



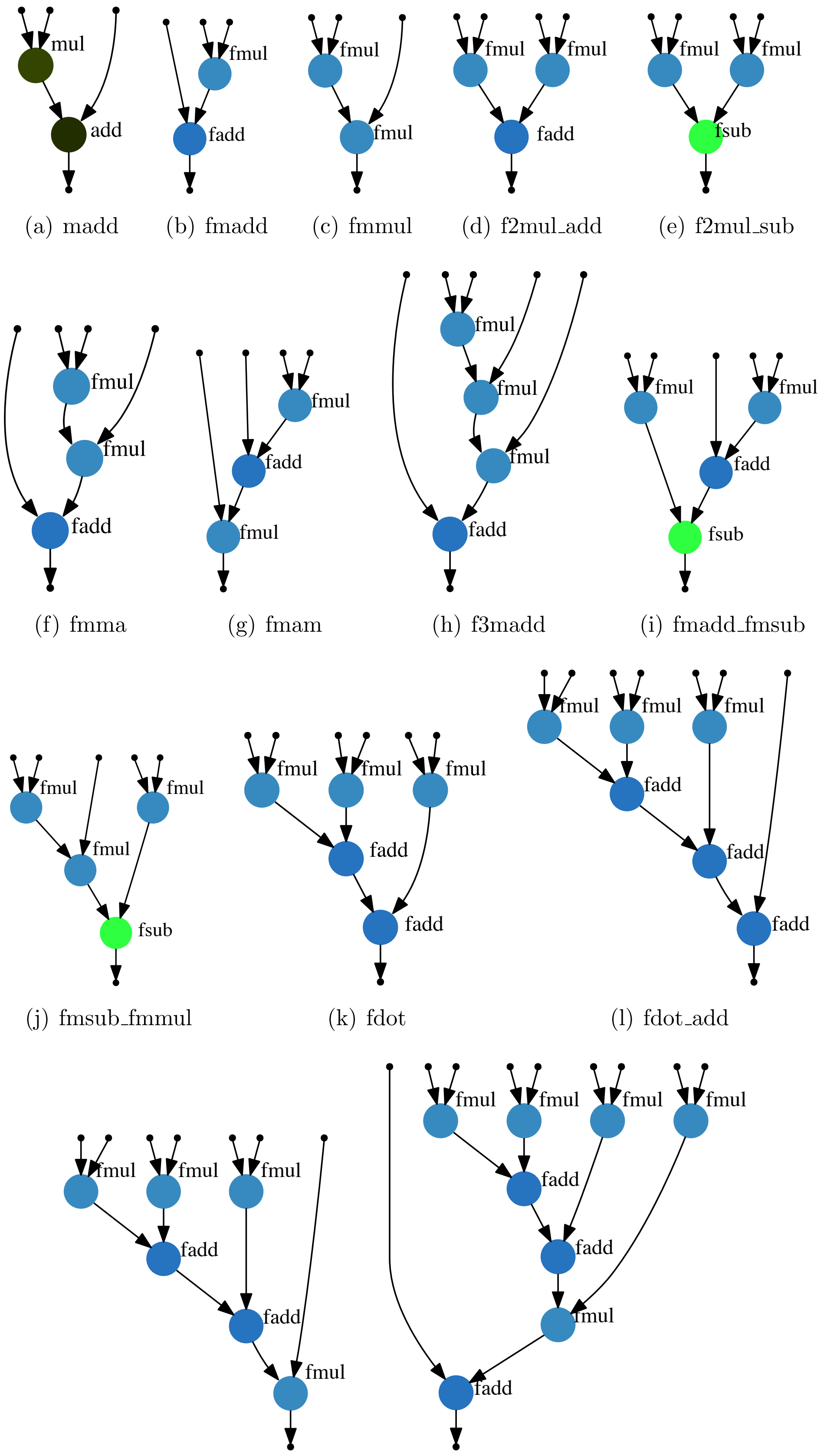
(a) xor_and

Figure A.5: CRC custom instructions.



(a) i2fpmul (b) fpadd2i (c) i_f_sma

Figure A.6: EDGE custom instructions.



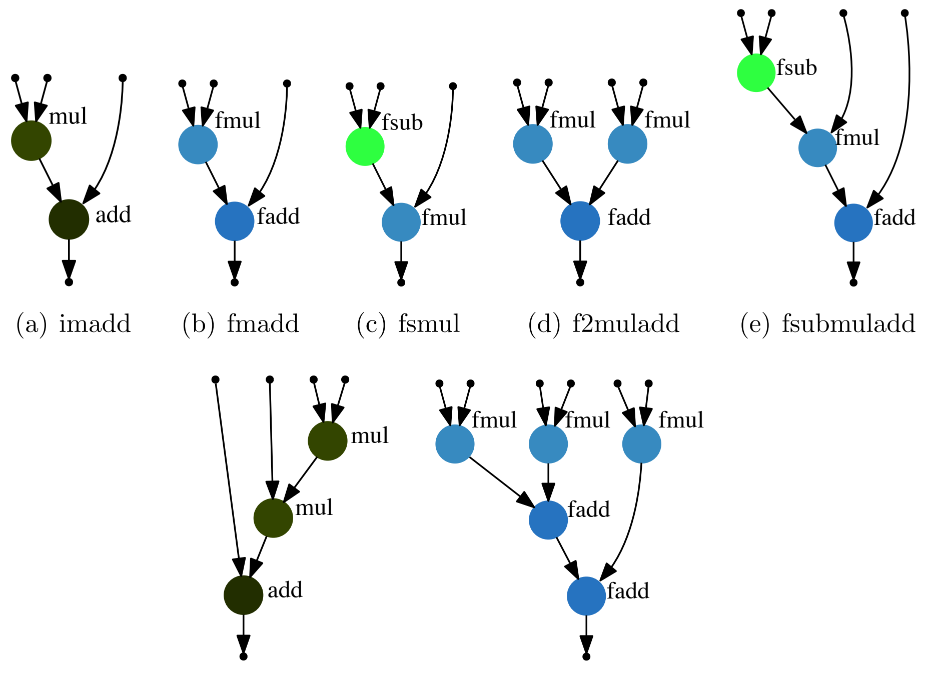
(a) madd (b) fmadd (c) fmmul (d) f2mul_add (e) f2mul_sub

(f) fmma (g) fmam (h) f3madd (i) fmadd_fmsub

(j) fmsub_fmmul (k) fdot (l) fdot_add

(m) fdot_mul (n) fdot2

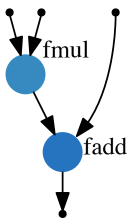
Figure A.7: RT custom instructions.



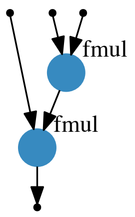
(a) imadd (b) fmadd (c) fsmul (d) f2muladd (e) fsubmuladd

(f) immula (g) fdot

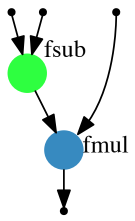
Figure A.8: VRC custom instructions.



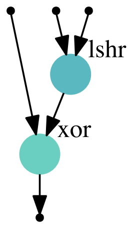
(a) fmadd_small



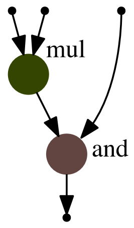
(b) fmmul_small



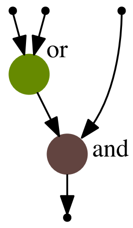
(c) fsubm_small



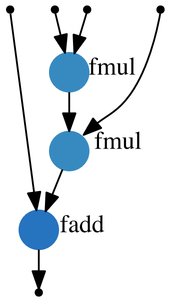
(d) lshr_xor_small



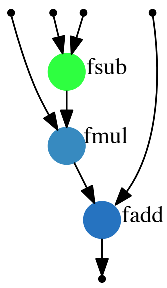
(e) mand_small



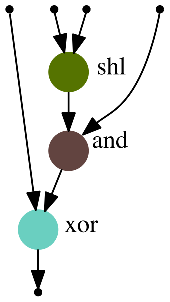
(f) or_and_small



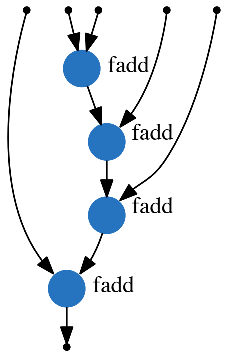
(g) fmmadd_small



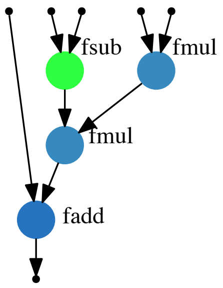
(h) fsub_madd_small



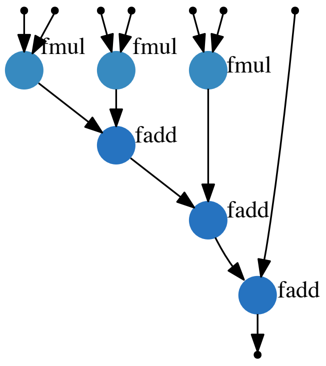
(i) shl_and_xor_small



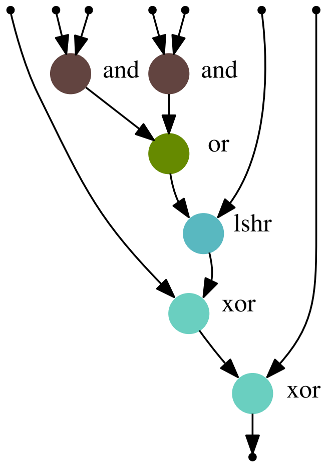
(j) f4add_small



(k) sp4



(l) fmul3_3add



(m) c6

Figure A.9: MPSO custom instructions.

Appendix B

Library of merged custom instructions

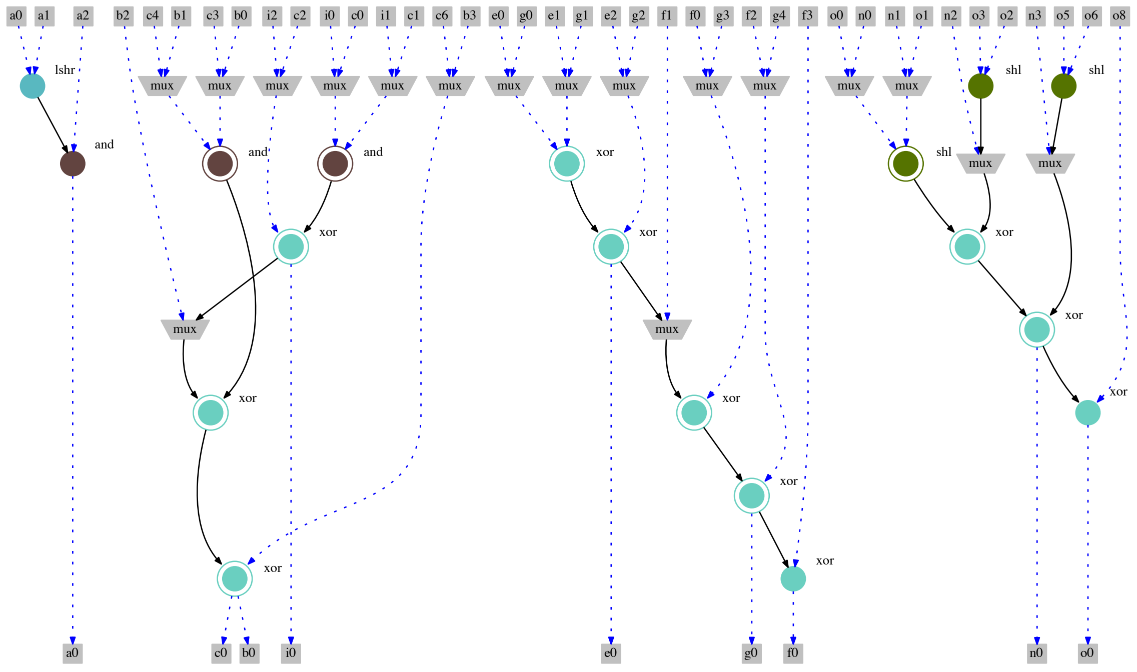


Figure B.1: Merged AES custom instructions.

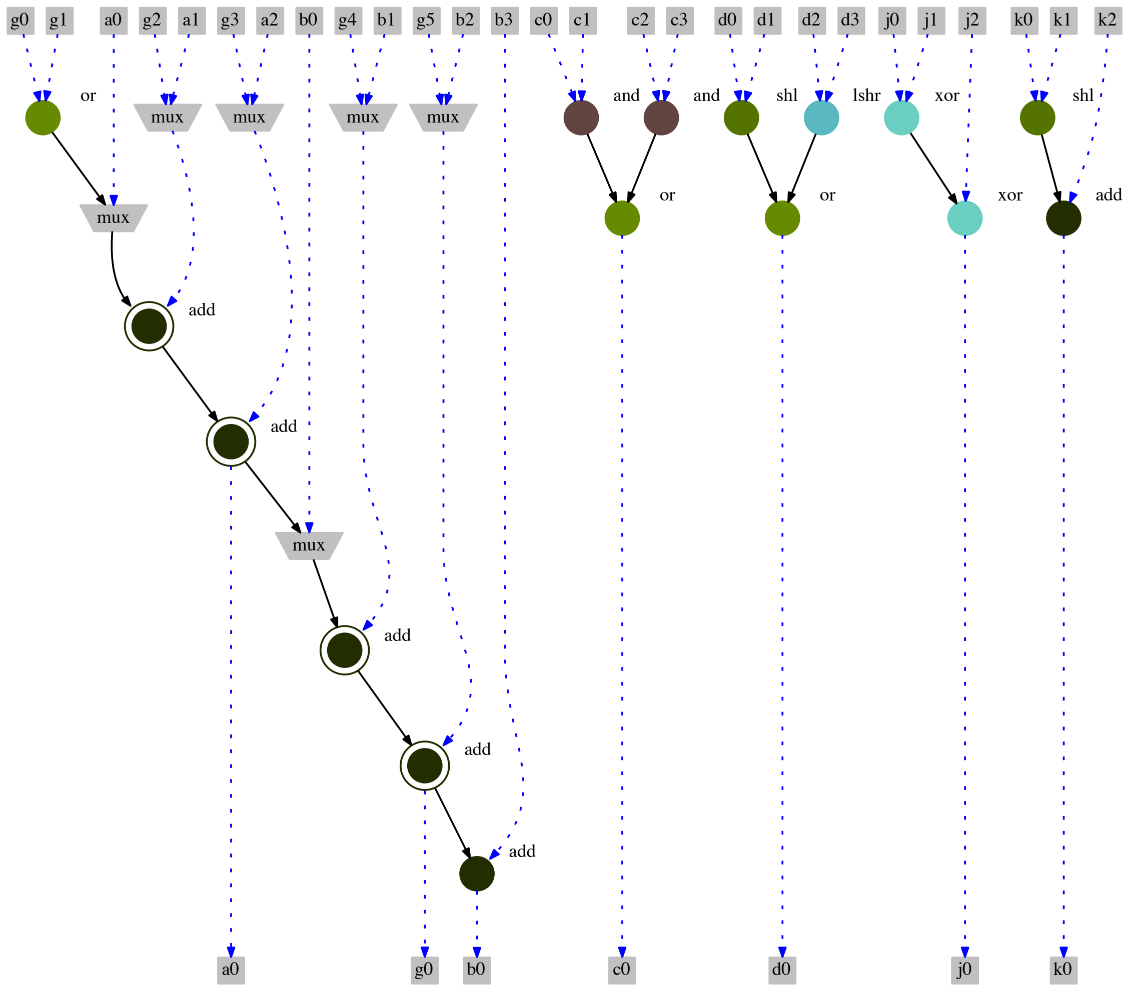


Figure B.2: Merged SHA custom instructions.

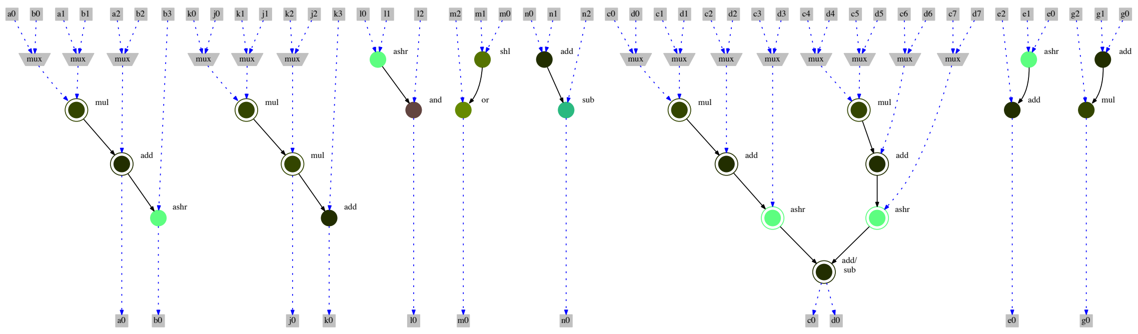

a0 b0 a1 b1 a2 b2 b3 k0 j0 k1 j1 k2 j2 k3 l0 l1 l2 m2 m1 m0 n0 n1 n2 c0 d0 c1 d1 c2 d2 c3 d3 c4 d4 c5 d5 c6 d6 c7 d7 e2 e1 e0 g2 g1 g0
mux
ashr
shl
add
ashr
add
mul
and
or
sub
mul
add
mul
add
add
ashr
add
ashr
ashr
add/
sub
a0 b0 j0 k0 l0 m0 n0 c0 d0 e0 g0

Figure B.3: Merged JPEG custom instructions.

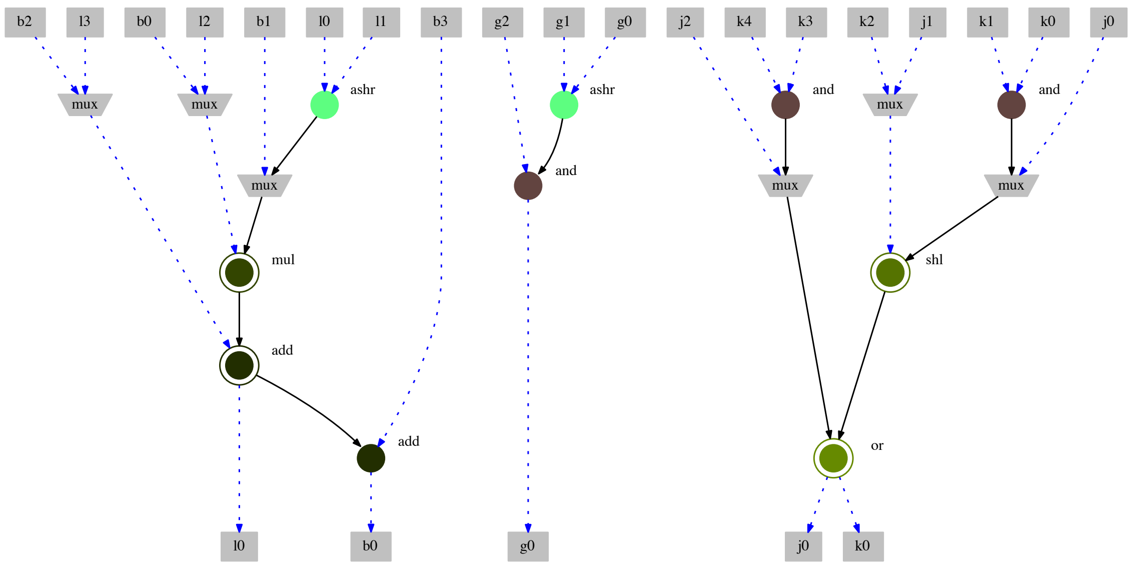


Figure B.4: Merged MJPEG custom instructions.

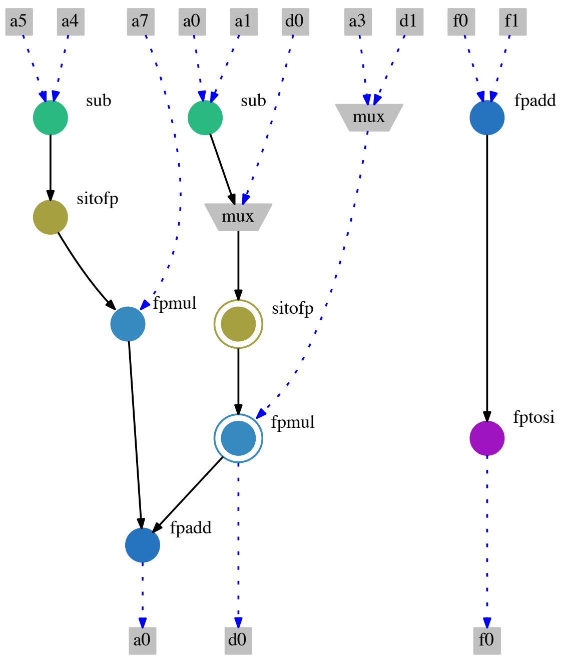


Figure B.5: Merged EDGE custom instructions.

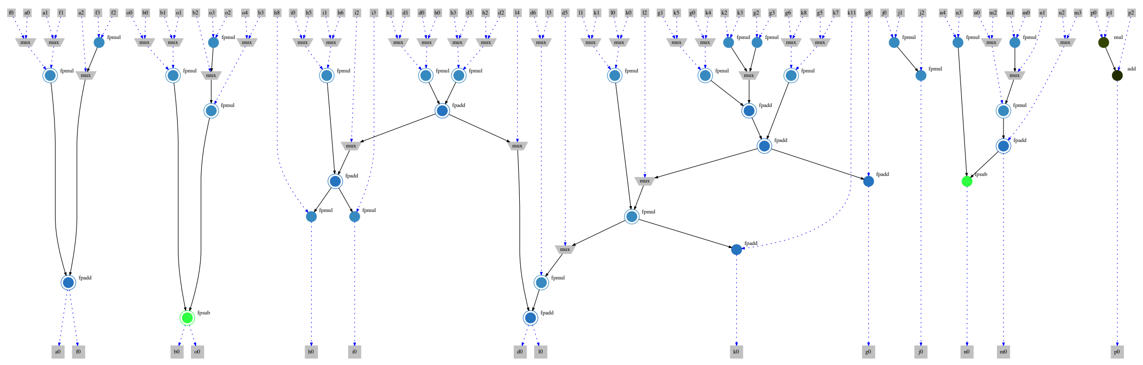


Figure B.6: Merged RT custom instructions.

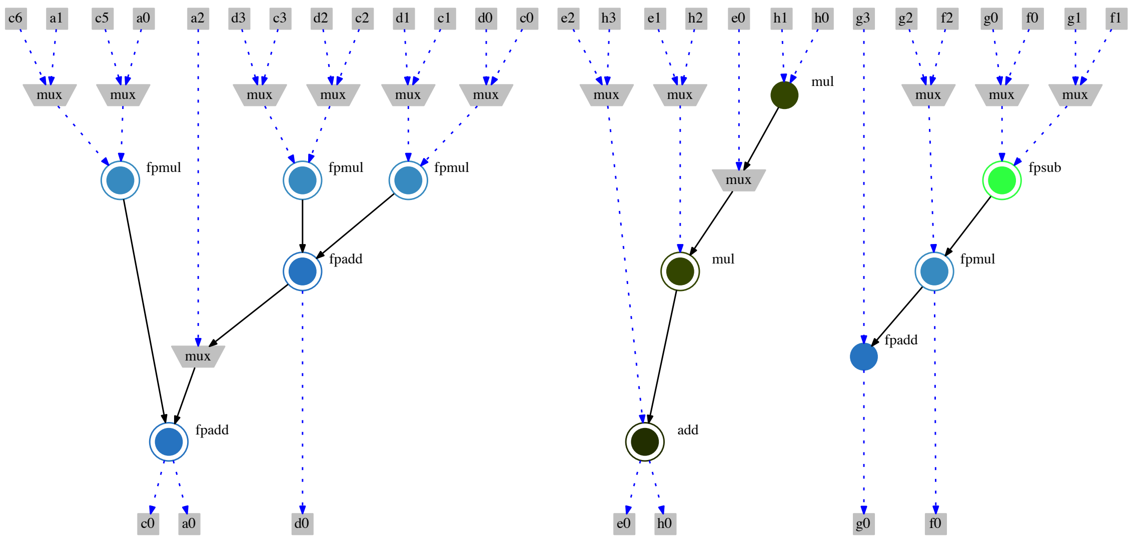


Figure B.7: Merged VRC custom instructions.

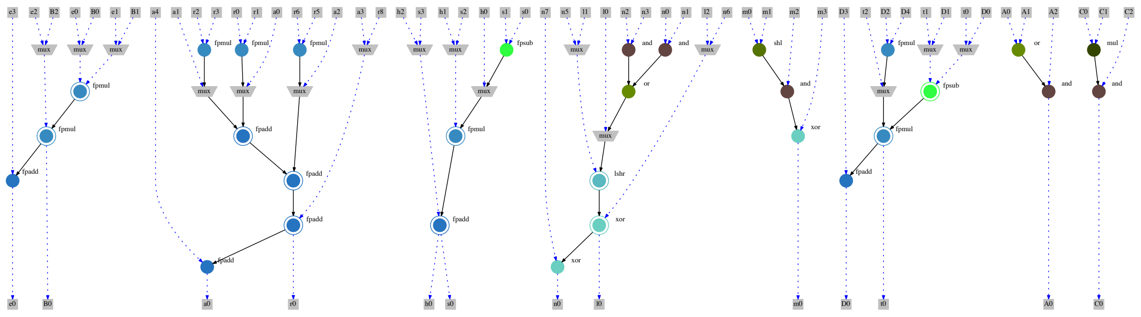
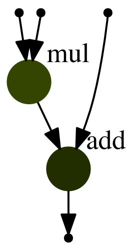


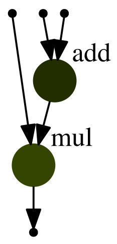
Figure B.8: Merged MPSO custom instructions.

Appendix C

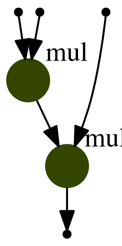
MPSoC custom instructions



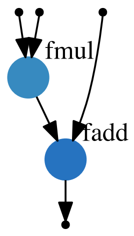
(a) imadd



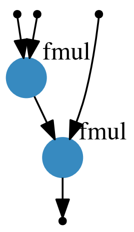
(b) iamul



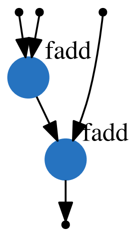
(c) immul



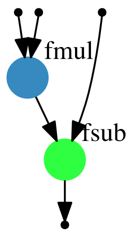
(d) fmadd



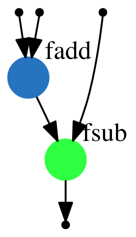
(e) fmmul



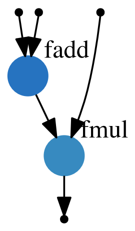
(f) faddadd



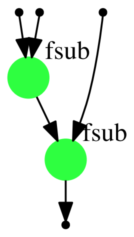
(g) fmsub



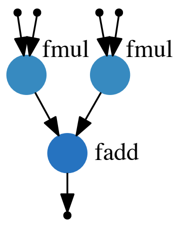
(h) faddsub



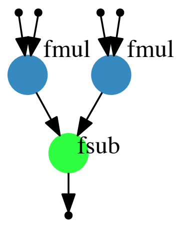
(i) faddmul



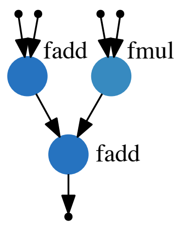
(j) fsubsub



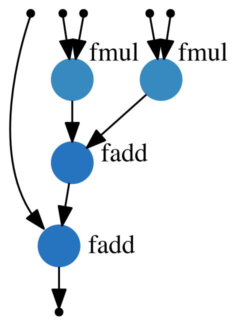
(k) f2mul_add



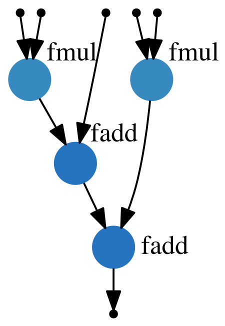
(l) f2mul_sub



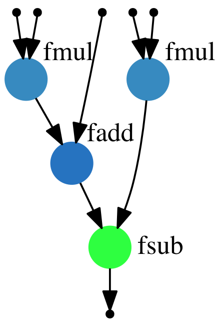
(m) famuladd



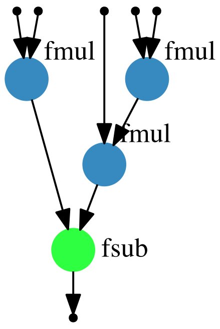
(n) f2mul_2add



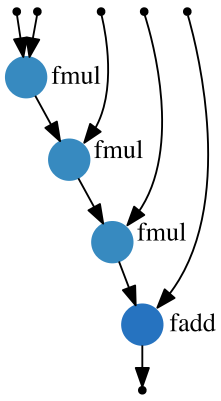
(o) fmadd_madd



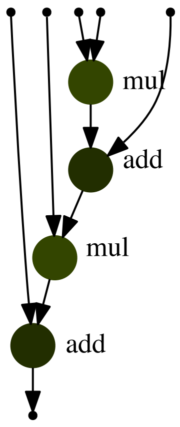
(p) fmadd_msub



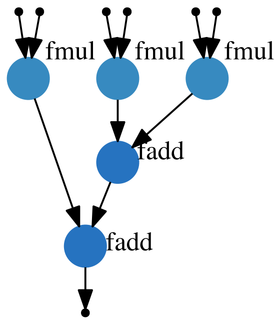
(q) fmsub_mmul



(r) f3mul_add



(s) imadd_madd



(t) fdot

Figure C.1: MPSoC-RT custom instructions.

Appendix D

List of publications

D.1 Journal Articles

1. NERY, A. S., JÓŹWIAK, L., LINDWER, M., et al. "Hardware Reuse in Modern Application-specific Processors and Accelerators", Microprocessors and Microsystems, v. 37, n. 6–7, pp. 684–692, ago. 2013.
2. NERY, A. S., NEDJAH, N., FRANÇA, F. M. G., et al. "Parallel Processing of Intersections for Ray-tracing in Application-specific Processors and GPGPUs", Microprocessors and Microsystems, v. 37, n. 6-7, pp. 739–749, ago. 2013.
3. NERY, A., NEDJAH, N., FRANÇA, F.M.G. "An efficient parallel architecture for ray-tracing". Analog Integrated Circuits and Signal Processing, v. 69, p. 1–14, 2012.
4. NERY, A. S., NEDJAH, N., FRANÇA, F. M. G. "Efficient hardware implementation of Ray Tracing based on an embedded software for intersection computation". Journal of Systems Architecture, v. i, p. 1–10, 2011.

D.2 In Conference Proceedings

1. NERY, A., NEDJAH, N., FRANÇA, F., et al. "A Framework for Automatic Custom Instruction Identification on Multi-Issue ASIPs". In: Industrial Informatics (INDIN), 12th IEEE International Conference on, pp. 428–433, June 2014.
2. NERY, A., NEDJAH, N., FRANÇA, F., et al. "Automatic complex instruction identification for efficient application mapping onto ASIPs". In: Circuits and Systems (LASCAS), 2014 IEEE 5th Latin American Symposium on, pp. 1–4, Feb 2014.
3. NERY, A., NEDJAH, N., FRANÇA, F., JÓŹWIAK, L. "Interactive Volume Rendering Based on Ray-Casting for Multi-core Architectures". In: Dayd, M., Marques, O., Nakajima, K. (Eds.), High Performance Computing for Computational Science - VECPAR 2012, v. 7851, Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 177–186, 2013.
4. NERY, A., NEDJAH, N., FRANÇA, F., et al. "A Reconfigurable Ray-Tracing Multi-Processor SoC with Hardware Replication-Aware Instruction Set Extension". In: Kolodziej, J., Di Martino, B., Talia, D., et al. (Eds.), Algorithms and Architectures for Parallel

Processing, v. 8285, Lecture Notes in Computer Science, Springer International Publishing, pp. 346–356, 2013.

5. NERY, A., NEDJAH, N., FRANÇA, JÓŹWIAK, L. “Massively Parallel Identification of Intersection Points for GPGPU Ray Tracing”. In: International Conference on Algorithms and Architectures for Parallel Processing, 2011, Melbourne. ICA3PP 2011 Workshops. Verlag Berlin Heidelberg: Springer, 2011. v. 2. p. 14–23.

6. NERY, A. S., NEDJAH, N., FRANÇA, F. M. G. “A Parallel Architecture for Ray-Tracing with an Embedded Intersection Algorithm”. In: International Symposium on Circuits and Systems, 2011, Rio de Janeiro. Proceedings of ISCAS 2011. Los Alamitos: IEEE Computer Society Press, 2011. p. 1491–1494.

7. NERY, A., NEDJAH, N., FRANÇA, JÓŹWIAK, L. “A Parallel Ray Tracing Architecture Suitable for Application-Specific Hardware and GPGPU Implementations”. In: Euromicro Conference on Digital System Design (DSD 2011), 2011, Oulu. Euromicro Conference on Digital System Design (DSD 2011). Los Alamitos, CA.: IEEE Press, 2011. v. 1.

8. NERY, A., JÓŹWIAK, L., LINDWER, M., et al. “Hardware Reuse in Modern Application-Specific Processors and Accelerators”. In: Digital System Design (DSD 2011), 2011 14th Euromicro Conference on, pp. 140–147, 31 2011-sept. 2 2011.