CUSTOMIZING THE COMPUTATION CAPABILITIES OF MICROPROCESSORS

by

Nathan T. Clark

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2007

Doctoral Committee:

Associate Professor Scott Mahlke, Chair
Professor Trevor N. Mudge
Associate Professor Todd M. Austin
Associate Professor Serap Savari
Krisztián Flautner, ARM Limited

ACKNOWLEDGEMENTS

First, I'd like to express sincerest gratitude to my adviser Scott Mahlke. I consider myself truly lucky to have worked with him these past years. He has shown incredible patience, served as an excellent mentor, and provided me every opportunity to succeed in this field.

I also owe thanks to the remaining members of my dissertation committee, Prof. Austin, Prof. Mudge, Kris Flautner, and Prof. Savari. They all donated their time to help shape this research into what it has become today. I would particularly like to thank Prof. Austin for convincing me to go to graduate school in the first place, and Kris Flautner for giving me several opportunities to apply this research in industry.

All of the research in this dissertation utilizes a software infrastructure too large for any one person to manage; none of this work would have been possible without the technical support of everyone in the CCCP research group. Hongtao Zhong wrote the first incarnation of the pattern matching code used in Chapter 2. Mike Chu and Jay Blome spent significant effort with me porting our compiler to the ARM instruction set. All of our porting efforts were based upon code generation work done by Rajiv Ravindran. Manjunath Kudlur and Hyunchul Park assisted me in getting a simulator with trace cache working, and more recently, Amir Hormati spent many hours on simulator hacking and developing compiler analysis tools for cyclic computation. Kevin Fan also helped me innumerable times, fixing the really hard bugs no one else wanted to touch. I feel very fortunate to have worked with a group of people so willing to help one another out, and so wiling to engage in technical discussions to help flesh-out ideas.

More importantly than the technical assistance, I'd like to thank all the members of the CCCP research group who I've ever shared an office with over the years for the social

support: Jay Blome, Mike Chu, Kevin Fan, Manjunath Kudlur, Steve Lieberman, Mojtaba Mehrara, Pracheeti Nagarkar, Hyunchul Park, Rajiv Ravindran, Misha Smelyanskiy, and Hongtao Zhong. You folks made coming to work a lot more fun, and I'd never have made it through without you. You can be certain that I'll be monitoring the "quote of the day" web page in anticipation of your future shenanigans.

Along that note, I really appreciate all the time spent with all the other friends I've made in grad. school here at Michigan. CSEG Drunks, you guys kept me sane.

Finally and most importantly, my family deserves major gratitude. My parents, Tom and Deb, and my sister, Beth, provided their unconditional love and support through this whole process, even though they found it a bit odd that I was still in school as a 21st grader. And I really appreciate the love and support of my wife, Jenny. She showed incredible patience to stick around Ann Arbor for four years while I finished my Ph.D., and only rarely asked when I would finish :) Jenny is as responsible for this dissertation as I am (in a good way).

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

LIST OF ALGORITHMS

Algorithm

ABSTRACT

CUSTOMIZING THE COMPUTATION CAPABILITIES OF MICROPROCESSORS

by

Nathan T. Clark

Chair: Scott Mahlke

Designers of microprocessor-based systems must constantly improve performance and increase computational efficiency in their designs to create value. To this end, it is increasingly common to see computation accelerators in general-purpose processor designs. Computation accelerators collapse portions of an application's dataflow graph, reducing the critical path of computations, easing the burden on processor resources, and reducing energy consumption in systems. There are many problems associated with adding accelerators to microprocessors, though. Design of accelerators, architectural integration, and software support all present major challenges.

This dissertation tackles these challenges in the context of accelerators targeting acyclic and cyclic patterns of computation. First, a technique to identify critical computation subgraphs within an application set is presented. This technique is hardware-cognizant and effectively generates a set of instruction set extensions given a domain of target applications. Next, several general-purpose accelerator structures are quantitatively designed using critical subgraph analysis for a broad application set.

The next challenge is architectural integration of accelerators. Traditionally, software invokes accelerators by statically encoding new instructions into the application binary. This is incredibly costly, though, requiring many portions of hardware and software to be redesigned. This dissertation develops strategies to utilize accelerators, without changing the instruction set. In the proposed approach, the microarchitecture translates applications at run-time, replacing computation subgraphs with microcode to utilize accelerators. We explore the tradeoffs in performing difficult aspects of the translation at compile-time, while retaining run-time replacement. This culminates in a simple microarchitectural interface that supports a plug-and-play model for integrating accelerators into a pre-designed microprocessor.

Software support is the last challenge in dealing with computation accelerators. The primary issue is difficulty in generating high-quality code utilizing accelerators. Hand-written assembly code is standard in industry, and if compiler support does exist, simple greedy algorithms are common. In this work, we investigate more thorough techniques for compiling for computation accelerators. Where greedy heuristics only explore one possible solution, the techniques in this dissertation explore the entire design space, when possible. Intelligent pruning methods ensure that compilation is both tractable and scalable.

CHAPTER 1

Introduction

For decades industry has produced, and consumers have relied on, exponential performance improvements from microprocessor systems. This continual performance improvement has enabled many applications, such as real-time ray tracing, that would have been computationally infeasible only a few years ago. Despite these advances, many very compelling application domains remain beyond the scope of everyday computer systems, so the quest for improving performance remains an important research goal.

The traditional method of performance improvement, through increased clock frequency, has fallen by the wayside as the increased power consumption now outweighs any performance benefits. This development has spurned a great deal of recent research in the area of multicore systems: trying to provide efficient performance improvement through increased parallelism.

Not all applications are well suited for multicore environments, though. In these situations, an increasingly popular way to efficiently provide more performance is through customized hardware, also known as computation accelerators. Adding accelerators to a general-purpose design not only provides significant performance improvements, but also major reductions in power consumption as well [120]. There are many examples of customized hardware being effectively used as part of a system-on-chip (SoC) in industry, for example the encryption coprocessor in Sun's UltraSPARC T2 [95].

There are three major challenges in utilizing hardware accelerators: design of the accelerators, cost effective architectural integration, and compilation support.

1.1 Design of Computation Accelerators

Efficiency versus programmability is the central tradeoff involved in designing hardware accelerators. Accelerators improve efficiency by performing larger amounts of computation in hardware than general-purpose designs. However, the more computation an accelerator performs in hardware, the less likely it is for that accelerator to be useful across multiple applications.

At present, the most popular strategy for exploiting computation accelerators is to build a system consisting of a number of special-purpose application specific integrated circuits, or ASICs, coupled with a general purpose processor. The ASICs are specially designed hardware accelerators to execute large portions of the application that would run too slowly if implemented on the processor. While this approach is effective, ASICs are costly to design and offer only a hardwired solution that permits almost no postprogrammability.

An alternative design strategy is to augment the core processor with small, acyclic special-purpose hardware to increase its computational capabilities in a cost-effective manner. The instruction set of the core processor is extended to feature an additional set of operations, and hardware support is added to execute these operations in the form of new function units. The Tensilica Xtensa is an example commercial effort in this area [49].

There are a number of benefits to augmenting the instruction set of a core processor with small acyclic accelerators. First, the system is postprogrammable and can tolerate changes to the application. Though the degree of application change is not arbitrary, the intent is that the customized processor should achieve similar performance levels with modest changes to the application, such as bug fixes or incremental modifications to a standard. Second, the computation intensive portions of applications from the same domain (e.g., encryption) are often similar in structure. Thus, the customized instructions can often be generalized in small ways to make them more useful across a set of applications. Last, some or all of the ASICs become unnecessary if the augmented core can achieve the desired level of performance. This lowers the cost of the system and the overall design time.

The central question with this approach is the degree of human effort required to design an efficient set of instruction set extensions. This effort can often be more time consuming

and expensive than the design of an ASIC. The current Xtensa system places much of this burden on the user to define, implement, and exploit the customized processor.

Automation is the key to making instruction set customization successful. To this end, this dissertation presents the design of a system that automates hardware selection of the acyclic custom instructions. Hardware design is accomplished via intelligent dataflow graph exploration. The exploration focuses on efficient discovery and selection of computation subgraphs from which custom hardware is constructed. The major challenge is navigating through an almost limitless design space without artificially constraining the size and shape of the subgraphs. This dissertation demonstrates that the technique is high quality, and provides a valuable tool for identifying critical computations in a target application set.

Once the acyclic accelerator design technique is in place, this work leverages it to further explore the programmability-efficiency tradeoff in different accelerator designs. Three novel, more general-purpose acyclic accelerators are presented: a configurable compute accelerator (CCA), a programmable carry function unit (PCFU), and a cyclic computation accelerator for loop bodies. The CCA and PCFU provide the functionality of a wide range of acyclic application-specific instruction set extensions in a single hardware unit. The CCA consists of an array of function units that can efficiently implement many common dataflow subgraphs. The PCFU implements these subgraphs using lookup-table based structures similar to a field-programmable gate array (FPGA). Both of these structures are more programmable than application- or domain-specific instruction set extensions; however, they are less efficient. At the other end of the spectrum is the loop accelerator. The loop accelerator only targets the innermost loops of applications, making it less programmable. However, since loops constitute larger pieces of computation than simple acyclic subgraphs, the loop accelerator is much more efficient.

1.2 Architectural Integration of Computation Accelerators

Hardware accelerators efficiently improve the performance of their targeted application domains, but they have problems associated with them, as well. The main problem is that there are significant non-recurring engineering costs associated with implementing accelerators. The addition of accelerators to a baseline processor brings along with it many of the issues associated with designing a brand new processor in the first place. For example, a new set of masks must be created to fabricate the chip, the chip must be reverified (using both functional and timing verification), and the new instructions must fit into a previously established pipeline timing model. Furthermore, applications must be re-engineered to incorporate support for the new accelerators.

To overcome these problems, this dissertation proposes a strategy to customize the computation capabilities of a processor within the context of a general-purpose instruction set, referred to as *transparent instruction set customization*. The goal is to extract many of the benefits of traditional hardware accelerators without having to break open the processor design and application binary each time. The fundamental idea is that subgraphs to be accelerated are identified and then dynamically replaced with microarchitectural instructions that configure and utilize whatever accelerators are present in the system.

Several different strategies are proposed for using these accelerators without changing the instruction set. One strategy, a fully dynamic scheme, performs subgraph identification and instruction replacement in hardware. This technique is effective for preexisting program binaries. To reduce hardware complexity, a hybrid static-dynamic strategy is proposed, which performs subgraph identification offline during the compilation process. Subgraphs that are to be mapped onto the accelerator are marked in the program binary to facilitate simple configuration and replacement at run-time by a hardware translator or dynamic compiler.

These utilization techniques culminate as an architectural framework to efficiently support transparent instruction set customization in a general-purpose processor. The framework utilizes a hybrid approach of statically-identified, dynamically-realized custom in-

structions. Subgraphs targeted for acceleration are identified during compilation or as a post-link optimization and are marked in the program executable. At run time, subgraphs are discovered, mapped, and executed on hardware accelerators. The hybrid approach enables the combination of sophisticated offline subgraph detection algorithms with the flexibility of online realization of the customized instructions. The key idea is that in order to facilitate efficient dynamic realization, the most difficult aspects of the translation problem should be performed statically.

Transparent instruction set customization is technique flexible enough to enable binary-compatible accelerator utilization for a wide range of accelerator designs. This dissertation demonstrates its application for the CCA, PCFU, loop accelerator, as well as single-instruction multiple-data (SIMD) accelerators.

1.3 Compilation for Computation Accelerators

An overlooked challenge with exploiting computation accelerators, and another focus of this dissertation, is the associated compiler support for accelerators. The compiler has two major tasks. First, it must identify candidate subgraphs in the target application that are functionally executable on the accelerator. This is essentially a subgraph isomorphism problem. The second task is to select which candidate subgraphs to actually execute on the computation accelerator. Candidates often overlap, thus the compiler must select a subset to maximize performance gain. This task is essentially a graph covering problem.

Most prior solutions employ a greedy compiler approach for both subgraph identification and selection to make the problem tractable. As with all greedy approaches, this approach can achieve sub-optimal solutions in both identification and selection. This dissertation proposes an new approach for compiler subgraph mapping that combines much more thorough methods with a set of intelligent pruning techniques. Pruning ensures the proposed algorithms are scalable in both application and accelerator size to provide practical compilation times.

1.4 Organization

The remainder of this dissertation is organized as follows. Chapter 2 develops an automated technique for designing high quality acyclic instruction set extensions for one or more target applications. This provides a useful tool for automatically extracting the critical computation subgraphs from a set of applications. Chapter 3 utilizes this technique to determine functionality requirements for a general-purpose acyclic subgraph accelerator. These requirements lead to the design of two novel families of accelerators based on arrays of combinational logic and lookup-tables. Following that, Chapter 4 investigates transparent instruction set customization: methods for utilizing accelerators without costly changes to a processor's instruction set. Chapter 4 introduces these ideas in the context of acyclic computation accelerators, and Chapter 6 extends transparent customization for SIMD accelerators. Turning attention to the compiler side, Chapter 5 develops automated methods for identifying computation subgraphs to map onto accelerators. The design and utilization of cyclic accelerators is covered in Chapter 7. Finally, Chapter 8 summarizes the results presented in this dissertation.

CHAPTER 2

Automatic Design of Domain-Specific Acyclic Accelerators

2.1 Introduction

In recent years, the markets for cellular phones, digital cameras, network routers, and other high performance but special purpose devices have grown explosively. In these systems, application-specific hardware design is used to meet the challenging cost, performance, and power demands. The most popular strategy is to build a system consisting of a number of highly specialized application specific integrated circuits (ASICs) coupled with a low cost core processor, such as an ARM [115]. The ASICs are specially designed hardware accelerators to execute the computationally demanding portions of the application that would run too slowly if implemented on the core processor. While this approach is effective, ASICs are costly to design and offer only a hardwired solution that permits almost no postprogrammability.

An alternative design strategy is to augment the core processor with special-purpose hardware to increase its computational capabilities in a cost-effective manner. The instruction set of the core processor is extended to feature an additional set of operations. Hardware support is added to execute these operations in the form of new function units or co-processor subsystems. The Tensilica Xtensa is an example commercial effort in this area [49].

There are a number of benefits to augmenting the instruction set of a core processor. First, the system is postprogrammable and can tolerate changes to the application. Though

the degree of application change is not arbitrary, the intent is that the customized processor should achieve similar performance levels with modest changes to the application, such as bug fixes or incremental modifications to a standard. Second, the computationally intensive portions of applications from the same domain (e.g., encryption) are often similar in structure. Thus, the customized instructions can often be generalized in small ways to make their use have applicability across a set of applications. Last, some or all of the ASICs become unnecessary if the augmented core can achieve the desired level of performance. This lowers the cost of the system and the overall design time.

One central question with this approach is the degree of human effort required to identify and implement an efficient set of instruction set extensions. In addition, the effort required to modify the software development tool chain to effectively understand the extended instruction set is substantial. This effort can often be more time consuming and expensive than the design of an ASIC.

We believe automation is the key to making instruction set customization successful. To this end, this chapter presents the design of a system that combines automatic hardware selection and seamless compiler exploitation of the custom instructions. Hardware design is accomplished via intelligent dataflow graph exploration. The exploration subsystem focuses on efficient discovery and selection of computation subgraphs from which custom hardware is constructed. The major challenge is navigating through an almost limitless design space without artificially constraining the size and shape of the subgraphs.

Once the custom instructions are discovered, several generalization techniques are applied to allow for quality mapping of subgraphs to each hardware unit. This ensures that custom instructions are useful across an entire domain of applications. These generalization techniques are unique to the field of instruction set customization.

Compiler exploitation of the custom instructions is accomplished through a flexible subgraph matching engine. Applications are analyzed to match computation subgraphs that can be replaced by custom instructions. This allows the customized hardware to be effectively utilized no matter what application is run on it. The compiler work presented here is later extended and more fully evaluated in Chapter 5.

Many other researchers have proposed systems to accomplish the task of automated in-

struction set generation. The contributions of this chapter are four fold. First, we present a novel technique for efficient dataflow graph exploration and selection. Second, we present the design and demonstrate the implementation of a complete system, including retargetable compiler. Most previous work neglects the problem of compiling to a processor with custom instructions. Third, and most importantly, we use the system to analyze how effectively instruction set extensions designed for one application can be applied to other applications in the same domain. Several techniques to increase the cross-application utility are explored. Lastly, we provide some analysis on how custom instructions differ when designed with multiple applications in mind.

2.2 Related Work

A large body of research has gone into instruction set customization. Work in [8], [109], [125], [54], [52], and [126] all showed possible gains from using this technique. While these works show the potential utility of instruction set customization, they do not provide methods to automate the process. Many other systems have been proposed to automate this process, though. These systems can be categorized based on how they solve two sub-problems: identification of custom instruction candidates and how to make use of the candidates.

Candidate Discovery - Informally stated, candidate discovery is determining subsets of an application's dataflow graph, or *DFG*, that would be amenable for implementation in hardware. In the most general sense, each node of the DFG can either be included or excluded from a candidate, yielding O($2^{numberofnodes}$) potential candidates. Several techniques have been proposed to handle the intractability of this problem.

Early work [4] side-stepped the candidate discovery problem altogether by predefining a set of candidates. This strategy requires a designer to enumerate a superset of useful candidates to select from, and utilizes design automation in the selection phase. While some advantages of customization are realized, this approach is limited by the large amount of work necessary to define an appropriate superset of candidates and the poor results obtained when an appropriate superset is not available.

Work by Bennett [13] proposes iterative combination of primitives that occur in subsequent lines of code to reduce static code size. This method assumes that a base instruction set is given corresponding to a high level language. Statistics are gathered on the frequency of operations occurring near each other and the highest ranking combination is chosen as a new instruction. This technique is irrespective of the dataflow graph and is primarily used as a code size reduction technique.

Bennett's work is similar to candidate discovery algorithms in [104], [103], [128], [11], [20], and [64], in that all of these papers propose iterative combination of primitives. Iterative solutions typically combine two nodes, replace all such occurrences in the DFG, and repeat until some constraints are met. These solutions have the benefit of very good run times (typically $O(N^2)$) when compared to more thorough strategies, but risk being stuck in local maxima. Each edge is combined in a locally optimal manner, reminiscent of greedy heuristics.

Holmer proposed a more global technique [56], which was later extended by [61]. This technique discovered candidates by performing an initial grouping of nodes based on the schedule time in the DFG, then iteratively breaking and recombining these groups. Work by Bose [16], is similar to this, except that this work operated on a syntax tree, instead of a DFG, and used many more candidate transformations than breaking and combining. Another major difference is that Holmer guided use of the transformations using simulated annealing, attempting to maximize the worth of the instruction set, where Bose performed transformations unguided with the expected goal of improvement. The application of these two algorithms was mainly for designing entire instruction sets, as opposed to just ISA extensions.

Choi [24] generated initial candidates in a similar manner to Holmer. This work advocated combining instructions that could be executed in parallel and then combining those parallel sets to create custom instructions that were both wide and deep. In order to cut down on the number of potential candidates explored, Choi used an artificial limit on how deep the combined instructions can be. The main contribution of [24] is a new formulation of the candidate discovery problem: they discovered candidates using a modification of the subset-sum problem, and attempted to find the minimal set of instruction extensions

to meet a certain performance requirement (as opposed to simply discovering the optimal instruction extensions for a given cost). The main weaknesses of this work are the artificial limit on custom instruction length and the initial phase of combining parallel instructions performed when it is not clear that parallel combination is best.

Other work proposes dealing with intractability by limiting the size of the problem. The algorithm proposed in [7] and [34] searches a full binary tree, where each step decides whether or not to include a node of the DFG into a candidate. Ways to prune the tree are proposed, helping avoid the worst case $O(2^N)$ runtime, but the size of the DFG must still be relatively constrained in order for the algorithm to complete in a timely manner. This limits its usefulness for very large basic blocks.

Some researches have proposed heuristic ways to limit the search space without artificially constraining it. In [6], the least used half of all candidates are removed after each iteration of candidate discovery. While this technique will catch all important candidates in hot portions of the code, it potentially misses useful candidates that are moderately used in many portions of the application. Work by Sun [119] performs a similar pruning, but uses a more complex priority function to rank the candidates, taking into account the number of inputs and outputs, as well as dynamic occurrences. In Sun's work, candidates that do not meet a certain percentage of the best discovered candidate so far are removed. Work in reconfigurable computing [98] initially partitions the DFG into small pieces based on heuristics. Candidates are then selected for these partitions. Heuristic based methods such as this have the benefit of not artificially constraining the problem by potentially getting stuck in local minima or limiting the types of candidates discovered.

Utilization - The problem of how to make use of the candidates is the other major issue to solve for instruction set customization. The vast majority of automated systems in this field have neglected this problem. Most systems combine the discovery and selection phases so that whenever candidates are selected, they are immediately replaced in the code, e.g. [61]. These systems typically do not provide methods to reuse the new instructions in other applications. As such, it is necessary to look at work in the compiler community.

Automated utilization of custom instructions generally happens during the code generation phase of compilation. Traditional code generation methods use a tree covering

approach [3] to map the DFG to an instruction set. The DFG is split into several trees, where each instruction in the ISA covers one or more nodes in the tree. The tree is covered using as few instructions as possible. The purpose behind splitting the DFG into trees is that there are linear time algorithms to optimally cover trees, making the process very quick.

One problem with this method, though, is that DFGs are not trees. It is shown in [77] that tree covering methods can yield suboptimal results, particularly in the presence of irregular subgraphs common in custom instructions. To overcome this, [77] proposes splitting all instructions into "register-transfer" primitives and recombining the primitives in an optimal manner using integer programming. Work by Liao [80] attacked the same problem, and developed an optimal solution for DFG covering by augmenting a binate covering formulation. While both of these solutions are optimal, they also have exponential runtime, and must be selectively used.

Research in [101] describes a new way to look at the code generation problem. In this work, computationally complex algorithms are used to insert custom instructions and heuristics handle the rest of code generation. An application is initially decomposed into an algebraic polynomial expression which is functionally equivalent to the original application. Next, the polynomial is manipulated symbolically in an attempt to use custom instructions as best as possible. For example, a polynomial could be expanded using function identities (e.g. adding 0 to a value) to better fit an existing custom instruction. Custom instructions are inserted as intrinsic function calls in the polynomial, and functionally equivalent high level language is output once all of this is complete. The high level language can then be used as input to a standard compiler. The main contribution of this work is the method of algebraically modifying of code to better make use of available instructions.

Our System - The candidate discovery technique proposed in this chapter is similar to the work in [7] in that a full exponential search is used where appropriate. The technique in this chapter differs in that it incorporates a heuristic function, similar to [119] and [98], to help divide the problem when exponential search is too slow. This is discussed in detail in Section 2.3. In Section 2.4, the custom instruction utilization framework is described. This framework ties together several ideas from other work into one system, and addresses some

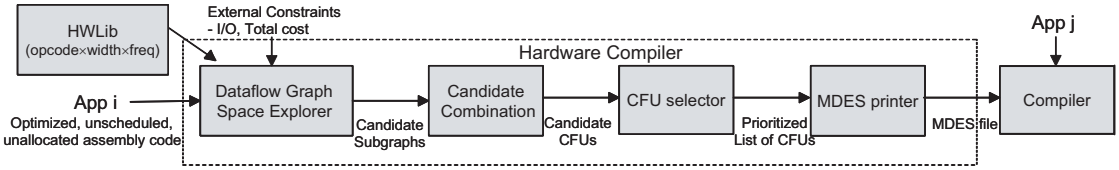


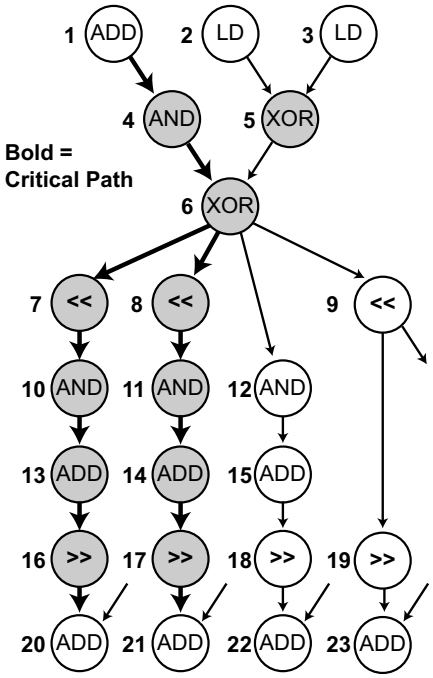
Figure 2.1: Organizational structure of the hardware compiler.

runtime issues with previously proposed solutions. The main contribution in this work is presented in Section 2.5: the custom instructions generated by our system are applied to benchmarks across several domains, and the results of these experiments are analyzed. Techniques to improve the effectiveness of cross domain instructions and the issue of how to design instructions for multiple applications are also tackled. Domain-wide discovery and analysis of custom instructions has not been previously examined.

2.3 Dataflow Graph Exploration

The purpose of the dataflow graph (DFG) explorer is to determine candidate subgraphs for instruction set extensions. Implementing subsets of the DFG in hardware typically allows for better performance, lower power consumption, and reduced code size than the corresponding implementation as a sequence of primitive operations. Determining which parts of a DFG would make the best custom instructions is a difficult problem, though. The most glaring difficulty is that the number of potential candidates for a given DFG increases exponentially in the number of operations. Exploration heuristics must be developed to overcome this problem.

The overall structure of our DFG exploration engine is shown in Figure 2.1. One or more applications are fed into the system as profiled assembly code. The code has not been scheduled and has not passed through register allocation, which is important so that false dependencies within the DFG are not created. Initially, the application passes through a DFG space explorer, which determines candidate subgraphs for potential instruction set extensions. The space explorer selects subgraphs subject to some externally defined constraints such as the maximum die area allowed for any custom function unit (CFU), or the



(a)

```
m ^= p [0];
r ^= p[ 1];
r ^= (((s[(m>>24L)] +
        s[0x0100+((m>>16L)&0xff)]) ^
        s[0x0200 + ((m>> 8L)&0xff)]) +
        s[0x0300+((m )&0xff)]) & 0xffffffff;
m ^= p[ 2];
m ^= (((s[(r>>24L)] +
         s[0x0100+((r>>16L)&0xff)]) ^
         s[0x0200+((r>> 8L)&0xff)]) +
         s[0x0300+((r )&0xff)]) & 0xffffffff;
```

(b)

Opcode	Adders	Cycles
+	1.0	0.30
AND	0.12	0.06
<<, >>*	0.01	~ 0
XOR	0.16	0.09

* These are shifts by compile time constants

(c)

Figure 2.2: A) Sample DFG from blowfish. Shaded nodes delineate a CFU. B) Preprocessed C code this DFG came from. C) Excerpt from the hardware library. “Adders” is the die area relative to a 32-bit adder.

maximum allowable register read and write ports. A hardware library provides timing and area estimates to the DFG explorer so that it can accurately gauge the cycle time and area requirements of combined primitive operations.

A list of subgraphs, annotated with area and timing estimates, is passed to a candidate combination stage. This stage groups subgraphs that would be executed on the same piece of hardware. Grouping the subgraphs creates a set of candidate CFUs and enables calculation of an estimated performance gain by using the profile weights of all the set members. The combination stage also performs some generalization steps to enable more subgraphs to map onto the same potential hardware implementations. All of this information is passed to a selection mechanism that determines which CFUs best meet the needs of the application(s). Finally, the prioritized list of CFUs is converted into a machine description (MDES) form that can be fed to the compiler.

Throughout this section, the DFG shown in Figure 2.2 from the *blowfish* applica-

tion [53] is used for illustrative purposes. For simplicity, each operation or node is assumed to take 1 cycle to execute on the baseline processor.

2.3.1 Subgraph Discovery

The exploration strategy employed in this work starts by examining each node in the DFG and using it as a seed for a candidate subgraph. This seed is grown downward along dataflow edges to create a new candidate. For example, if the seed was node 6 in Figure 2.2, it would be grown to nodes 7, 8, 9, and 12. When candidates overlap with each other (such as the candidates with nodes 6-7 and 6-8 in the example), a new candidate is created with the union of their nodes (6-7-8). During growth, each intermediate candidate is recorded for potential implementation as a CFU. Growing the candidates continues until some external constraints are met, such as the candidate crossing a control flow boundary or exceeding the number of register read ports available.

Initially, this system used a naïve implementation that looked at all possible dataflow edges to grow the seed nodes. Using this approach guarantees identification of the best possible set of connected candidates, since all possible candidates are generated. However, the number of candidates quickly grows out of control for many applications.

The key observation gained from experimenting with this naïve approach is that growing along the majority of dataflow edges examined by exponential growth simply do not make sense. For example, assuming the goal is maximizing performance on the DFG in Figure 2.2, growing along the edge between nodes 6 and 9 has little value, because node 9 is not on the critical path (i.e., the longest dependence path(s) in the DFG).

Previous work [7] [34] has shown that using an exponential solution, such as growing along all edges is sufficiently fast for some applications, when intelligent pruning is used. There are many applications that have too many nodes for exponential search, though. For these large DFGs, we propose using a *guide function* to determine which edges are directions that do not need to be grown toward. By heuristically removing unimportant edges, the DFG is effectively partitioned into smaller sections, which can then be used by the exponential growth algorithm described above. This strategy allows us to maintain the

quality of the resultant candidates without taking exponential time or resources.

Our previous work [32] proposed using a guide function as a method for inclusion, which is to say that only edges which were determined to be important were grown along. This effectively avoided the exponential search associated with subgraph discovery, but in some instances overcompensated for the problem. Using the guide function to remove edges takes the opposite approach and only prunes the search space when necessary.

One important characteristic of this technique is that the partitioning step can be tuned to more efficiently explore the design space. Partitioning the DFG into just a few larger sections will ensure better coverage of the design space, while partitioning the DFG into many smaller sections will result in reduced run times and memory consumption of the exploration algorithm. An example of exploiting this tradeoff would be using larger partitions in parts of the DFG that have higher profile weight, as they are more likely to yield important candidates. All previously proposed solutions use a single exploration strategy for all parts of the DFG, where as this technique can modify its strategy to effectively use the computational resources available.

2.3.2 Guide Function

The purpose of the guide function is to intelligently rank which dataflow edges would most likely be involved in unimportant candidates. The guide function essentially tries to replace the architect by determining these unimportant edges, thus its decisions must reflect the same properties the architect would use. The guide function proposed here uses three categories to rank the desirability of edges: criticality, latency, and area. In the candidate discovery system, each of the guide function categories is allotted 10 points of weight, and the sum of these categories determines the total desirability of each edge. The edges with the lowest desirability are more likely to be cut if the DFG needs to be partitioned.

Criticality - This category ranks edges highly when they appear on the longest dependence path(s) of a DFG. CFUs that occur on the critical path are likely to give the application performance improvement, because they shrink the height of the DFG. Since performance improvement is typically the most desired result of CFUs, cutting edges along

these paths is undesirable. An example of this from Figure 2.2 would be the edges from node 6. The edges toward nodes 7 or 8 would rank higher in terms of criticality than would the direction toward nodes 9 or 12, because the aforementioned nodes are on the critical path. Points are awarded using the equation $\frac{10}{slack+1}$, where slack is the number of cycles an operation can be delayed without lengthening the critical path. Thus, the edge from node 6 to 7 would get $\frac{10}{0+1} = 10$ points and the edge from node 6 to 9 would get $\frac{10}{2+1} = 3.33$ points. Note that it is important to give candidate edges credit even when they are slightly off the critical path as the heuristic provides because auxiliary paths often become critical after several CFUs are formed.

Latency - Combining operations that require fewer cycles to execute in conjunction than they do individually will lead to high quality CFU candidates. The largest performance gains are possible by combining several low latency operations, such as bit-wise logicals, where many can be executed in a single cycle. Conversely, if two nodes on an edge cannot be executed in fewer cycles when combined, then the resultant candidate is less beneficial. The latency category models this trend. Latency points are distributed using the equation $\frac{old\ latency}{new\ latency} * 10$. The latency of a CFU is calculated by summing up the fractional delays of the two atomic operations (see Figure 2.2c) connected to the edge. For example, node 10 can be executed in 0.06 cycles as indicated by the 'Cycles' entry for the AND opcode in Figure 2.2c. Exploring the edge toward node 13, which has a latency of 0.3 cycles, would get $\frac{0.06}{0.06+0.30} * 10 = 1.67$ points. In contrast, growing from node 6 toward node 9, would get nearly all ($\frac{0.09}{0.09+0} * 10 = 10$) the points allotted for latency.

Area - Since cost is a major constraint in the design of embedded processors, area is an important factor in the choice of CFUs. The guide function considers the area to be the sum of the areas required to implement each primitive operation on an edge (see Figure 2.2c). Note that register file ports are a design constraint, thus they do not factor into the area calculation. Further, CFUs do require additional decode logic and interconnect, but we assume that primitive operation area is the dominant term. The area category gives more points to directions that least increase the total area of the candidate. Area points are calculated the same way as latency, $\frac{old\ area}{new\ area} * 10$. As an example, growing from node 19 to node 23 would yield $\frac{0.01}{1.0+0.01} * 10 = 0.1$ points and growing from 9 to 19 would yield

$\frac{0.01}{0.01+0.01} * 10 = 5$ points.

The guide function gives a weight to each edge in the DFG. If the DFG proves to be too large for the exponential exploration algorithm, a recursive bisection is performed on the DFG until the partitions are small enough (typically 50 or fewer nodes). For example, if the DFG in Figure 2.2 was too large, then it would have to be split into smaller pieces. To do that, at least one edge would have to be cut. In this figure, the edge from node 18 to node 22 is the first choice to cut, because it has the smallest weight according to the guide function. Cutting that edge creates two new partitions and eliminates all candidates that contain that edge from being explored. If the two new partitions are still two large, the process is repeated until they are small enough. The partitioning is performed using hMetis [63], a high-quality multi-level partitioner. Edge weights from the guide function lead hMetis toward cutting edges which will not lead to good candidates. In practice, partitioning the DFG greatly reduces the design space to the point where most applications can be fully explored in under 5 minutes on a Pentium 4 system.

With the guide function/partitioning heuristic in place, it is important to verify two points: that the heuristic does indeed prune the search space, and that good candidates are not missed because the partitioner incorrectly precludes them. Figure 2.3a demonstrates the first point. Each dot on this graph represents the number of candidates examined when exploring one basic block from three encryption applications that are characterized by large loop bodies. This figure shows that the partitioner is able to effectively curve the exponential growth associated with the DFG exploration problem. This algorithm can be used on very large DFGs and without artificially constraining the types of candidates generated, which are both weaknesses of some previously proposed algorithms.

To ensure that good candidates are not dismissed, the heuristic was compared against a full exponential search using strict external constraints (candidates were only allowed 3 input and 1 output port). Figure 2.3B shows the speedups[1] attained from using the candidates generated by both algorithms. As shown in the figure, the two curves track identically due to the fact that the partitioning heuristic did not prune any important candidates during its search. DFGs can be constructed where the heuristic will miss important candidates, but

[1]The experimental setup used to obtain this data is explained in detail in Section 2.5.

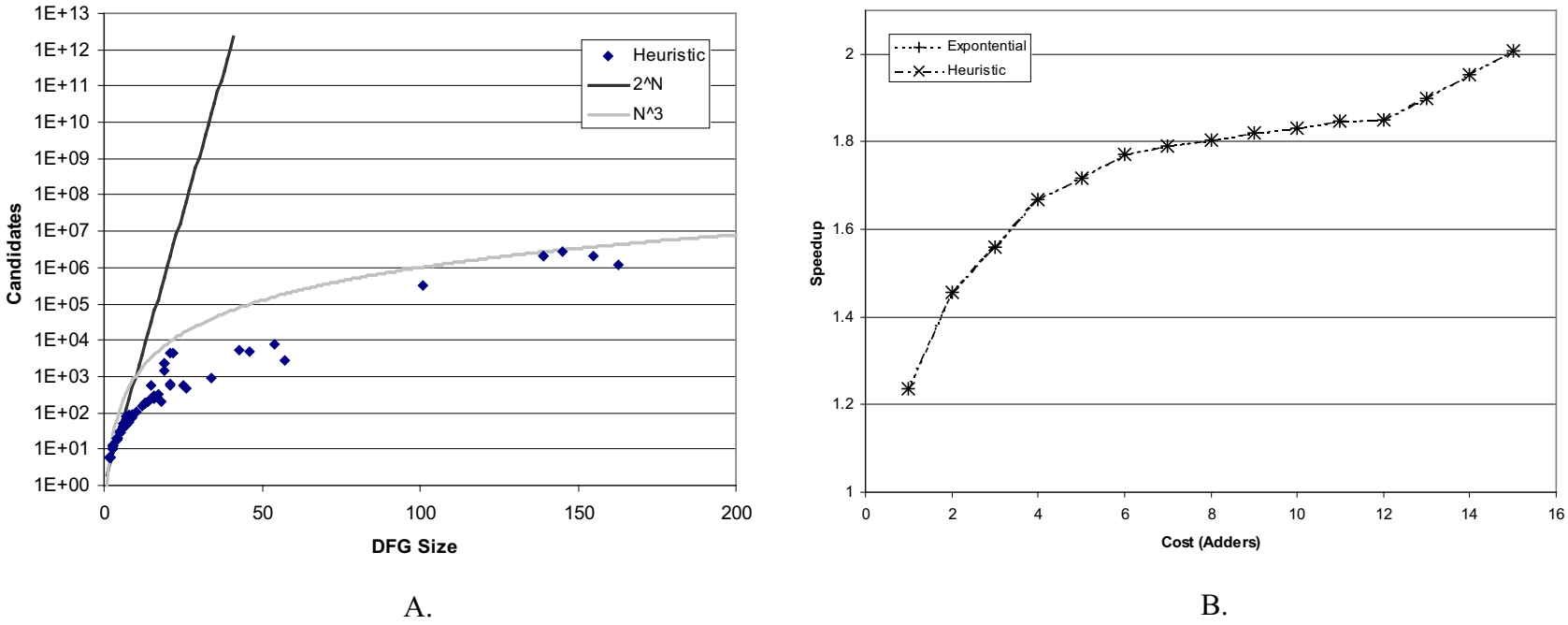


Figure 2.3: A) The number of candidates examined for DFGs in three encryption benchmarks. B) The average speedup on those benchmarks using full exponential and heuristic search techniques.

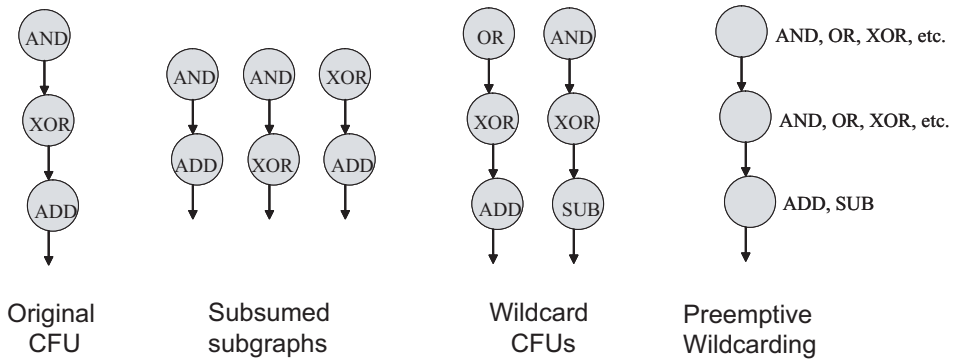


Figure 2.4: Examples of generalization techniques.

this was not observed in any of the test cases.

2.3.3 Candidate Combination and Generalization

After discovery, it is a straightforward process to group identical candidate subgraphs together into candidate CFUs. A simple test that checks graph equivalence, while taking into account commutativity, accomplishes this. For example, if subgraphs 7-10-13-16 and 8-11-14-17 were discovered in Figure 2.2, the graphs would be checked for equivalence and then combined into the candidate CFU "<<-AND-ADD->>". The profile weights are then used to get an estimate of the number of cycles each CFU improves performance. In the case that CFUs are being designed for multiple applications, the cycle estimates are scaled to ensure that one application does not dominate simply because it has a longer execution profile. Using a compiler to get an exact measurement by scheduling with each instruction is possible, but the complexity makes this solution undesirable. In practice, the profile-based estimates proved reasonably accurate.

After candidate grouping, a generalization process takes place to make the candidates more useful across a domain of applications. Two techniques are employed to accomplish this. The first is *subsumed subgraphs*. Subsumed subgraphs take advantage of the fact that most atomic operations have an associated identity input, allowing values to pass through a node without changing. Using Figure 2.4 as an example, if CFU "AND-XOR-ADD" was discovered, CFU "AND-ADD" can be executed on the same hardware because one input

of the XOR operation could be set to 0. CFUs “AND-XOR” and “XOR-ADD” could also be subsumed by “AND-XOR-ADD”. The cost of implementing these subsumed subgraphs is simply a MUX on one input of every node being bypassed; thus, for very little additional cost the number of subgraphs that map onto a CFU is increased.

The second generalization technique is called *wildcarding*. Wildcards are subgraphs that are a similar shape as the original CFU, but operations at one node may differ. Combining two CFUs with similar structure allows us to share hardware and map multiple subgraphs onto the same CFU. Two examples of wildcards are given in Figure 2.4. If the original CFU implements “AND-XOR-ADD”, then both “OR-XOR-ADD” and “AND-XOR-SUB” would be recorded as potential wildcards, if they appeared in the input DFG, since they only differ by one node from the original CFU.

A stronger version of wildcarding, termed *preemptive wildcarding* in this work, generalizes a CFU to have many potential operations at each node. Unlike regular wildcarding, the preemptive subgraphs do not necessarily have to appear in the input DFG. The idea behind preemptive wildcarding is that many operations have very similar hardware implementations, e.g. ADD and SUB, or can be added to a node for very little cost. Additionally, we have observed that applications within a domain have similar shaped DFGs, even if the operations at individual nodes do not match. Preemptively adding this functionality allows many more subgraphs to map to a single CFUs and makes them much more useful across a domain of applications. Again, an example of preemptive wildcarding is given in Figure 2.4. Here logical operations were added to the AND and XOR nodes, and SUB was added to the ADD node.

It is important to note that in this phase of the exploration framework, no binding decisions are made with regards to subsumed subgraphs and wildcarding. The generalization phase simply creates new, generalized candidates with updated area and cycle estimates to reflect what the potential cost and benefit of generalizing each original candidate. The selection algorithm can then weigh the costs and benefits of generalizing and make an appropriate decision. Since preemptive wildcarding often does not improve the estimated cycle savings for an input DFG, it is always performed if the cost required is less than a predefined threshold.

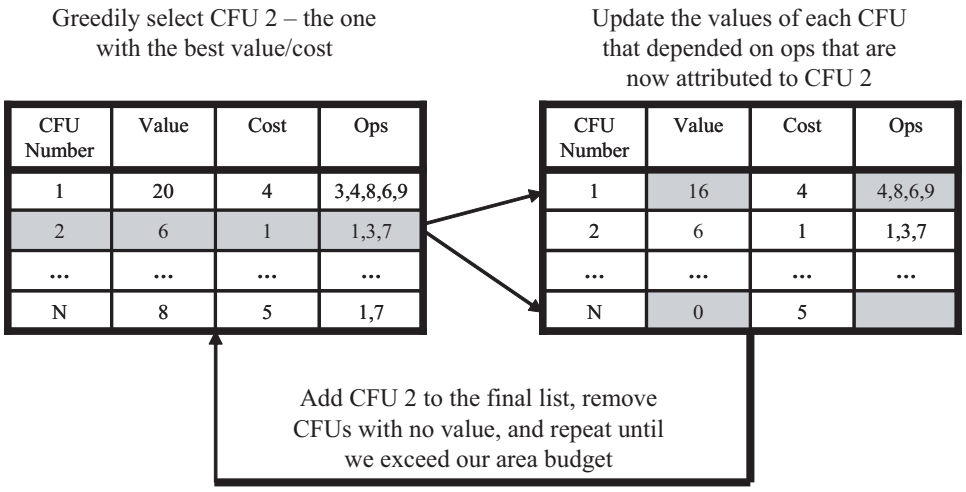


Figure 2.5: Greedy approach to CFU selection.

2.3.4 Candidate Selection

Selecting CFUs with a given area constraint is similar to the 0/1 knapsack problem. There is a set of resources (the CFUs) that all have a value (the estimated cycle savings) and a cost (die area), and the goal is to maximize the total value for a given cost. It is widely known that the 0/1 knapsack problem is NP-complete, although it is solvable in pseudo-polynomial time using dynamic programming. Strategies are needed to avoid intractability in this stage of design automation as well.

It is important to mention that CFU selection has one caveat missing in the 0/1 knapsack problem: the values of all the other CFUs change once a CFU is selected for inclusion. Individual operations can appear in multiple CFU candidates. Once a CFU is selected, it is necessary to update the estimated cycle savings of the other CFUs so that double counting does not occur. Using an example from Figure 2.2 again, assume the two highest ranked CFUs were 7-10-13-16, and 7-10-13. If 7-10-13-16 was selected first and did not update the value of 7-10-13 to reflect the fact that it can no longer use any of its operations, then 7-10-13 would be selected also, even though it would provide no gain above what 7-10-13-16 already provided.

One strategy used for CFU selection is a simple greedy method, illustrated in Figure

2.5. Given a list of CFU candidates, the one with the best ratio of $\frac{value}{cost}$ is greedily selected. Once CFU 2 is selected, the heuristic iterates through the list of remaining candidates and removes operations that were claimed by it. In Figure 2.5, operations 1 and 7 were removed from CFU N and its value was updated to 0, as it had no more operations left. Operation 3 was removed from CFU 1 and its value was likewise updated to 16. Once all CFUs are updated, the selection process is repeated until the area budget is exhausted.

Because the selection heuristic is greedy, it is not guaranteed to give an optimal solution, and frequently does not. For example, when the greedy algorithm selects based only on estimated cycle savings, performance does poorly at the low cost budget points compared to when it selects based on $\frac{value}{cost}$. However, the opposite is true at high cost points.

In an attempt to improve the selection heuristic, a version based on dynamic programming was implemented as well. This is a straightforward extension of the algorithms presented in [58]. The problem with the dynamic programming method is that it requires candidates to update their estimated value many more times than the greedy method. This computational overhead is quite significant, and in order to alleviate it, a simplifying assumption is made. Prior to the selection, each operation is assigned to the candidate with the largest estimated speedup. This eliminates the need to frequently update candidate values, but potentially misleads the selector. Despite this, the dynamic programming method typically provides better results than the greedy method.

Dealing with wildcards and subsumed subgraphs adds another challenge to the selection process. The main issue is the possibility that implementing a subsumed subgraph as a separate CFU is more desirable than implementing it on existing, subsuming hardware. As an example consider the large gray CFU from Figure 2.2. If "XOR - $<<$" were to be run on custom hardware, it could be done for a minimal area overhead on the large, gray CFU; however, there would be a latency penalty of going through three more operations (there are no early exits from operations 7 or 8). It may be that creating a special "XOR - $<<$" unit is the better solution. Subsumed CFUs are not removed from the selection pool, so that the option to include both the subsumed and subsuming candidate is available.

Another issue is whether to count all the subsumed subgraphs and wildcards when determining the estimated value of a CFU. If they are counted, then in addition to updating

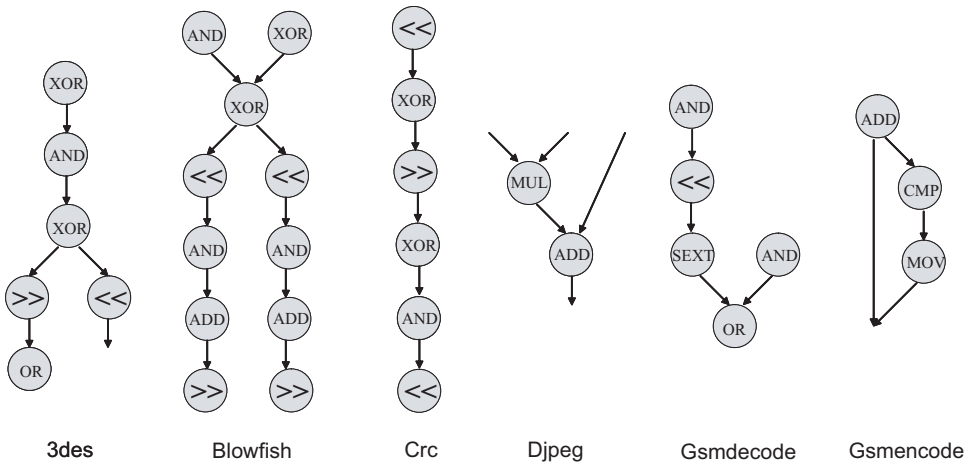


Figure 2.6: A selection of CFUs generated by the exploration system.

the estimated value of other CFUs based on the operations in the candidate subgraphs, it is also necessary to update the values based on all the operations in the subsumed or wildcard candidate subgraphs. This creates a large computational overhead for every subgraph selection. Additionally, this means frequently attributing operations to small subsumed portions of a large CFU, when more performance could have been gained by attributing them to a separate CFU (like the example in the previous paragraph). The case just described occurs quite frequently, so CFUs are selected as if they had no subsumed subgraphs or wildcards. When a selection is made, the costs of the subsumed subgraphs and wildcards are updated to reflect that they can now be added for very little cost overhead.

2.3.5 Example CFUs

The types of CFUs generated by this system are quite varied, depending on the input DFGs. Some examples of selected CFUs across six applications are shown in Figure 2.6. Often, the system generates instructions that an architect would expect to see, such as the multiply-accumulate selected for djpeg and the saturating-add selected for gsmencode. This provides some empirical evidence that the system is making intelligent decisions. The system also frequently generates custom instructions that appear very unusual, too, such as the ones for 3des, blowfish, crc, and gsmdecode.

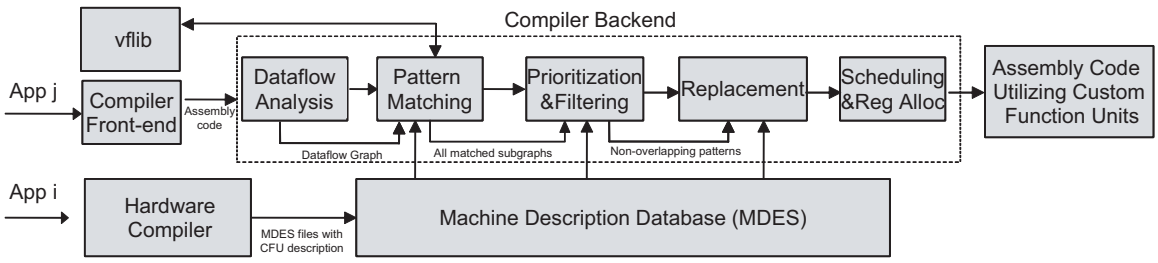


Figure 2.7: Organizational structure of a compiler supporting custom instructions.

2.4 Compiler Utilization

The purpose of the compiler is to automatically exploit CFUs available for any given application. The basic structure of the retargetable compiler is shown in Figure 2.7. Applications are run through a front-end, producing a generic RISC assembly code. The assembly code is unscheduled and uses virtual registers. The compiler uses a machine description, or MDES, to determine what CFUs are available for use. Given the assembly code and MDES, the compiler performs dataflow analysis to generate a DFG, discovers all subgraphs in the DFG that match available CFUs, prioritizes these matches, replaces the matches with custom instructions, and finally performs the typical tasks of register allocation and scheduling. The steps that differ from traditional compilation techniques are described in detail below. Again, the compilation strategy presented in this section is more fully described and evaluated in Chapter 5.

2.4.1 Pattern Matching

Pattern matching is the most critical step in CFU utilization. The first step in this process is determining all available CFUs from the MDES. From a high level, the MDES describes what resources a CFU consumes, the latency of the operation, the number and type of inputs and outputs, and the structure of the subgraph that the CFU implements.

Discovering the subgraphs in the DFG can be viewed as the subgraph isomorphism problem, which is known to be NP-complete. To perform subgraph identification, the

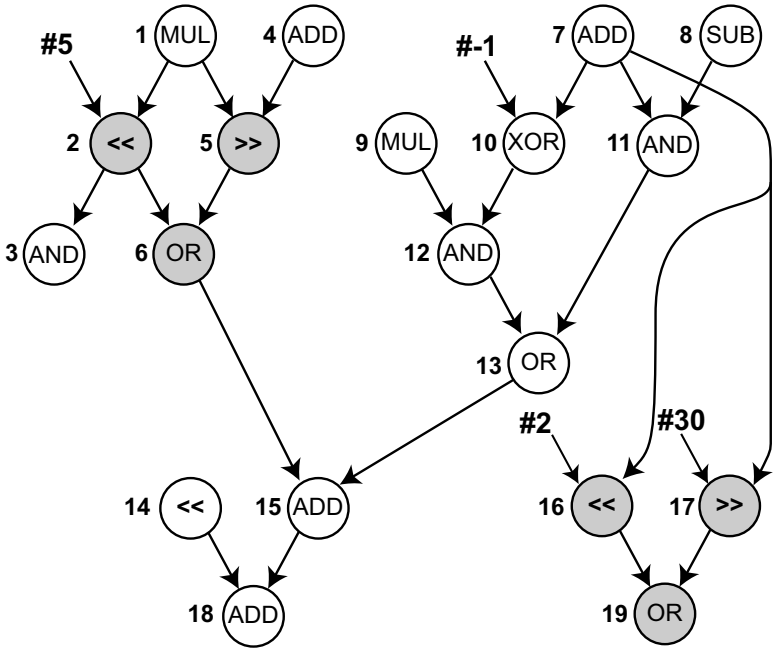


Figure 2.8: DFG similar to one from sha.

vflib graph matching library [35] is employed. While the algorithm used in vflib is still exponential worst case, the best case is only polynomial, and the overhead added to the compile time found in practice is minimal.

The vflib algorithm finds matching subgraphs by starting at individual nodes that occur both in the DFG and the CFU. These nodes are termed a *partial match*. The partial matches are then expanded along DFG edges to create new partial matches in a manner that is similar to DFG space exploration.

Figure 2.8 shows part of a DFG that is similar to one in the sha benchmark [53]. Given a CFU to implement the operations in subgraph 2-5-6, the pattern matcher would begin by looking at all left shift ($<<$) nodes: 2, 14, and 16. These partial matches would then be grown toward all consumers, since node 2 has a consumer in the CFU. This would create partial matches 2-3, 2-6, 14-18, and 16-19. 2-3 and 14-18 no longer match the CFU, so only 2-6 and 16-19 are considered. These two partial matches are then grown toward the producers of 6 and 19, since the original CFU had two producers feeding the OR node. This process continues until all the partial matches either definitively match or do not. Subgraph

matching is repeated for all CFUs, so that all potential subgraph matches in the DFG are discovered.

At this stage, the same operation may appear in multiple subgraph matches. Deciding which match an operation should be placed in is an NP-hard problem, though. The optimal solution proposed in [80] was prohibitively slow when implemented in our compiler. To overcome this, a partitioning technique was again employed.

The traditional solution mentioned previously uses a binate covering formulation which optimally maps CFUs onto a DFG. Typically an entire DFG is mapped at one time in this method. However, there are usually very few nodes that appear in multiple matches, meaning that the matches do not frequently overlap. This fact allows the problem to be separated into several, independent binate coverings on subsets of the DFG. To illustrate this, consider a mapping was being performed on the DFG in Figure 2.8. If no matches contained both nodes 13 and 15, then the graph could then be partitioned along the edge that connects them. Binate covering could be done independently on the left nodes and the right partitions without sacrificing optimality. A branch and bound algorithm was used to solve the binate covering formulation. Once the sub-problems are solved, an optimal solution to the entire DFG can be constructed from the optimal sub-solutions, much more quickly than when looking at the entire DFG at once. While cases can be constructed to make this technique prohibitively slow as well, in practice it was very fast, typically taking no more time than the scheduling phase of compilation.

2.4.2 Custom Instruction Replacement

On the surface, replacing the matched subgraph with a custom instruction is fairly simple. There are some important issues that must be considered in order to guarantee the correctness of the resultant program, however. Using the DFG shown in Figure 2.8, subgraph 2-5-6 will be replaced with a custom instruction. The question that arises is, “Where should the custom instruction be placed in relation to other operations in the assembly code?” To ensure correctness of the program, the custom instruction must be placed after all the predecessors of the operations in the subgraph (after nodes 1 and 4 in this example),

and also before all the successors (nodes 3 and 15 here). Assuming the node identifiers define the sequential order of the assembly code for this example, there is a potential problem with where to place the custom instruction. Replacing node 2 is incorrect because the custom instruction would be placed before node 4. Similarly, replacing nodes 5 or 6 is incorrect because it would be placed after node 3.

To prevent this from occurring, the assembly code is reorganized prior to subgraph replacement. For subgraph 2-5-6, the last scheduled predecessor is node 4 and the earliest scheduled successor is node 3. As long as the custom instruction is inserted between these operations, program semantics will be maintained. For every subgraph match, if the last predecessor comes after any successors, then those successors and any operations dependent them are moved after the last predecessor. In this example, we would move node 3 after node 4 , and then safely insert the custom instruction after the last predecessor.

Once the subgraphs are replaced and the code is reordered for correctness, scheduling and register allocation take place, leaving us with an application that intelligently utilizes the available CFUs.

2.5 Experimental Results

The system proposed was constructed as part of the Trimaran research infrastructure [121]. The DFG exploration engine was implemented as a standalone module, and the compiler backend was modified to facilitate subgraph matching and replacement. The cycle time and area estimates in the hardware library were calculated using Synopsis design tools and an Artisan 0.18μ standard cell library.

For this evaluation, two simplifying assumptions are made. First, no memory instructions were included in CFUs. Having custom instructions that access memory creates CFUs with non-deterministic latency as well as requires consideration of cache ports during DFG exploration. Memory disambiguation within a custom instruction must also be factored when doing pattern replacement in the compiler. The second assumption was that custom instructions were not allowed to contain branches or cross control flow boundaries (if-conversion of the code is allowed, however). These restrictions were put in place so

that custom instructions can remain stateless and atomic. Both assumptions are due to limitations in the DFG explorer and compiler, and do not reflect inherent limitations of the approach.

Sixteen full benchmarks were run through the CFU generation system and fifteen sets of CFUs for each benchmark were created. Each set corresponds to an area budget allotted to the CFUs (relative to one 32-bit ripple-carry adder, two adders, etc.). The sixteen benchmarks can be divided into four domains: encryption, network, audio, and image. The encryption category contains 5 benchmarks (blowfish, rijndael, and sha) from MiBench [53] and two other encryption applications (3des and Rc4). The network category consists of three benchmarks (crc, ipchains, and url) from NetBench [90], and the audio (gsmdecode, gsmencode, rawcaudio, and rawdaudio) and image (cjpeg, djpeg, epic, and mpeg2dec) domains are from MediaBench [75].

The baseline processor for the experiments is a four-wide VLIW that can issue one integer, one floating-point, one memory, and one branch instruction each cycle. The instruction set and latencies of each instruction are similar to those of the ARM-7 [115]. In all of our studies, the custom instructions require an integer issue slot to execute, thus an integer operation and a custom instruction cannot execute in the same cycle. This was done so that any speedups observed are due to custom instructions and not from adding parallelism to the processor. A 300 MHz system clock was assumed for timing constraints, and custom instructions that require more than one clock cycle to execute are pipelined so as not to affect cycle time. A maximum of four input and two output ports was placed as an external limit on all CFUs generated. Generally speaking, approximately 10-20 custom instructions were needed to attain the maximum speedups presented in the following figures. It is important that this number is small, in order to keep the impact on instruction set encoding minimal.

Although not presented in this work, a prototype of this system has been built in the ARM OptimoDE framework [28]. This prototype allowed us to measure the actual die area overhead for adding custom instructions to a processor. While the prototype implementation was fairly naïve, we found that custom instructions could be added to a processor for roughly 20% additional die area. The majority of this overhead was due to additional

control bits that resulted from adding an issue slot for custom instructions. Note that in our simulations, no issue slot is added, and thus the overhead for the model used in this chapter will likely be much less than the 20% reported in [28].

Performance Versus Area: The four graphs in Figure 2.9 compare the performance gain in each of the four benchmark domains as the total cost budget for CFUs is varied. Each line in the graphs represents the speedup of an application with CFUs designed specifically for it compared to the baseline processor. One of the interesting trends in these graphs is that speedups seen in benchmarks vary greatly. Encryption benchmarks tend to benefit quite a bit from CFUs, with 3des, rijndael, and sha showing speedups of 2.39, 2.08, and 1.91, respectively, at the higher cost points. On the contrary, some applications in other domains show very little speedup (e.g. mpeg2dec, epic, and ipchains). Investigation into this revealed that these benchmarks had a significant number of branches and memory operations, which hindered the combinable operations available for the DFG explorer. Conversely, the encryption benchmarks contained large subgraphs dominated by simple arithmetic and logical operations, which are ideally suited for custom hardware.

Another very noticeable trend in Figure 2.9 (blowfish and djpeg in particular) is that at some higher cost points there is a dip in speedup. This is due directly to the greedy node assignment in the dynamic programming selection heuristic. Recall that in the proposed selection algorithm (see Section 2.3.4), when a node appears in multiple candidates, a pre-selection pass removes that node from all candidates except the one with the largest estimated latency decrease. This assignment saves a great deal of computation during selection, but is just a heuristic, and can make bad decisions. For blowfish, a speedup of approximately 1.7 is attained at cost point 4, by assigning several nodes to small and generally useful CFUs. At cost point 5 the heuristic assigned the nodes that used to be in small CFUs to a very large CFU, artificially inflating its value in comparison to the smaller ones. In reality, the compiler was not able to make use of the large CFU as well as the smaller ones, and thus performance suffered.

Cross Compilation and Generalization: Figures 2.10 and 2.11 show the performance of applications when run with CFUs designed for other applications within the same domain. Two benchmarks are listed for each set of bars; the first one is the application being

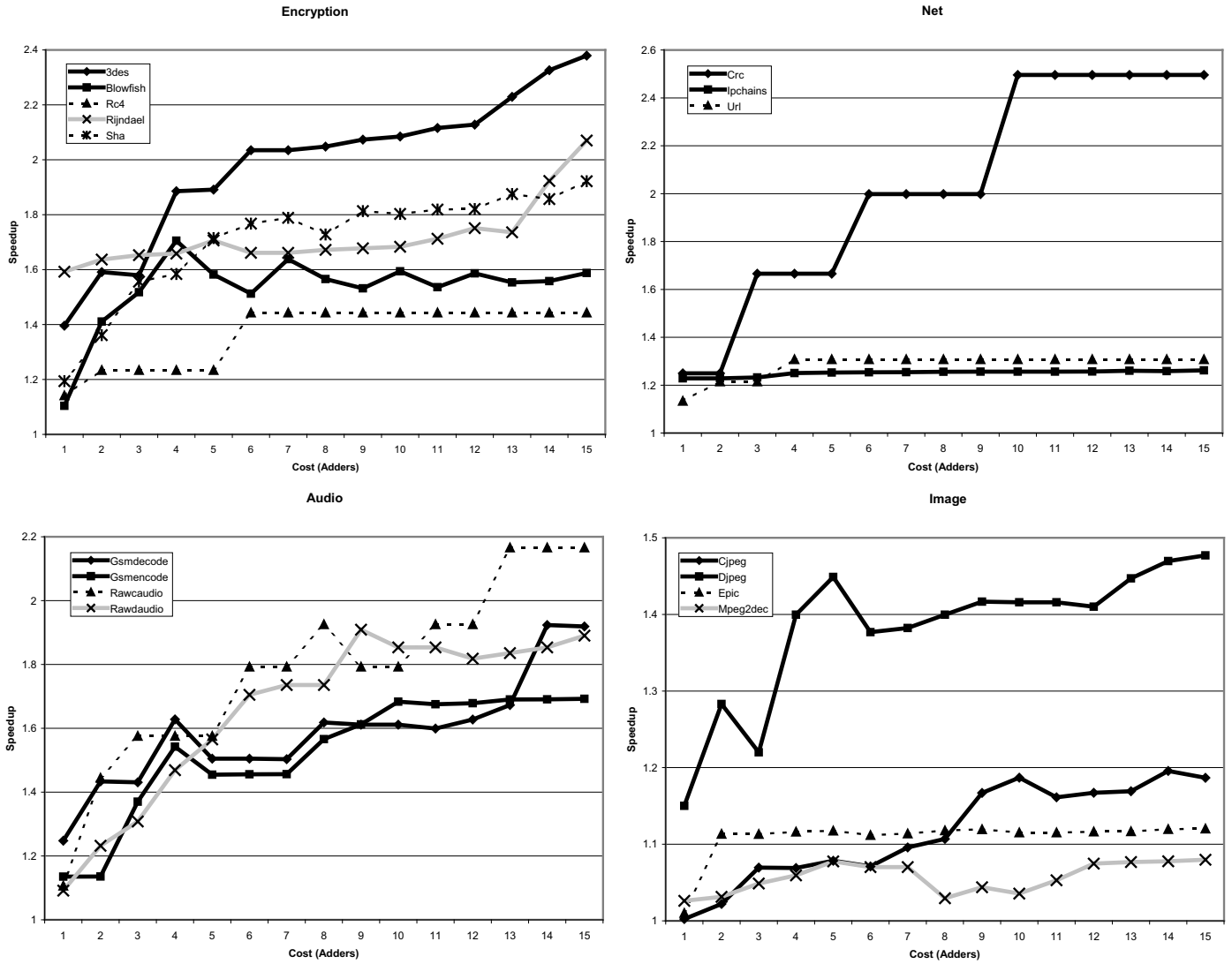


Figure 2.9: Performance of four application groups as CFU cost budget is increased from 1 to 15 32-bit adders.

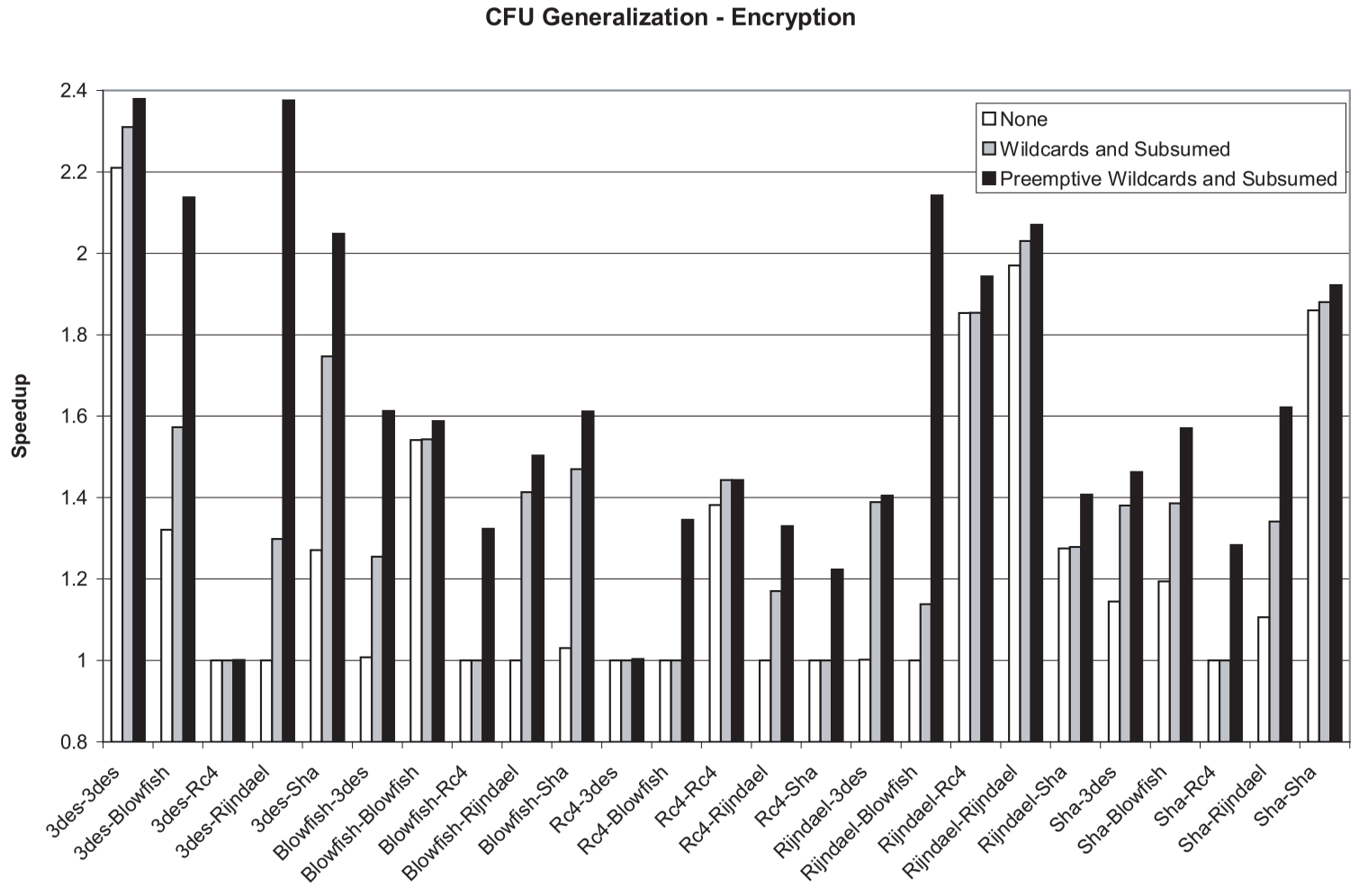


Figure 2.10: Effect of subsumed subgraphs and wildcards in Encryption at the 15-adder cost point.

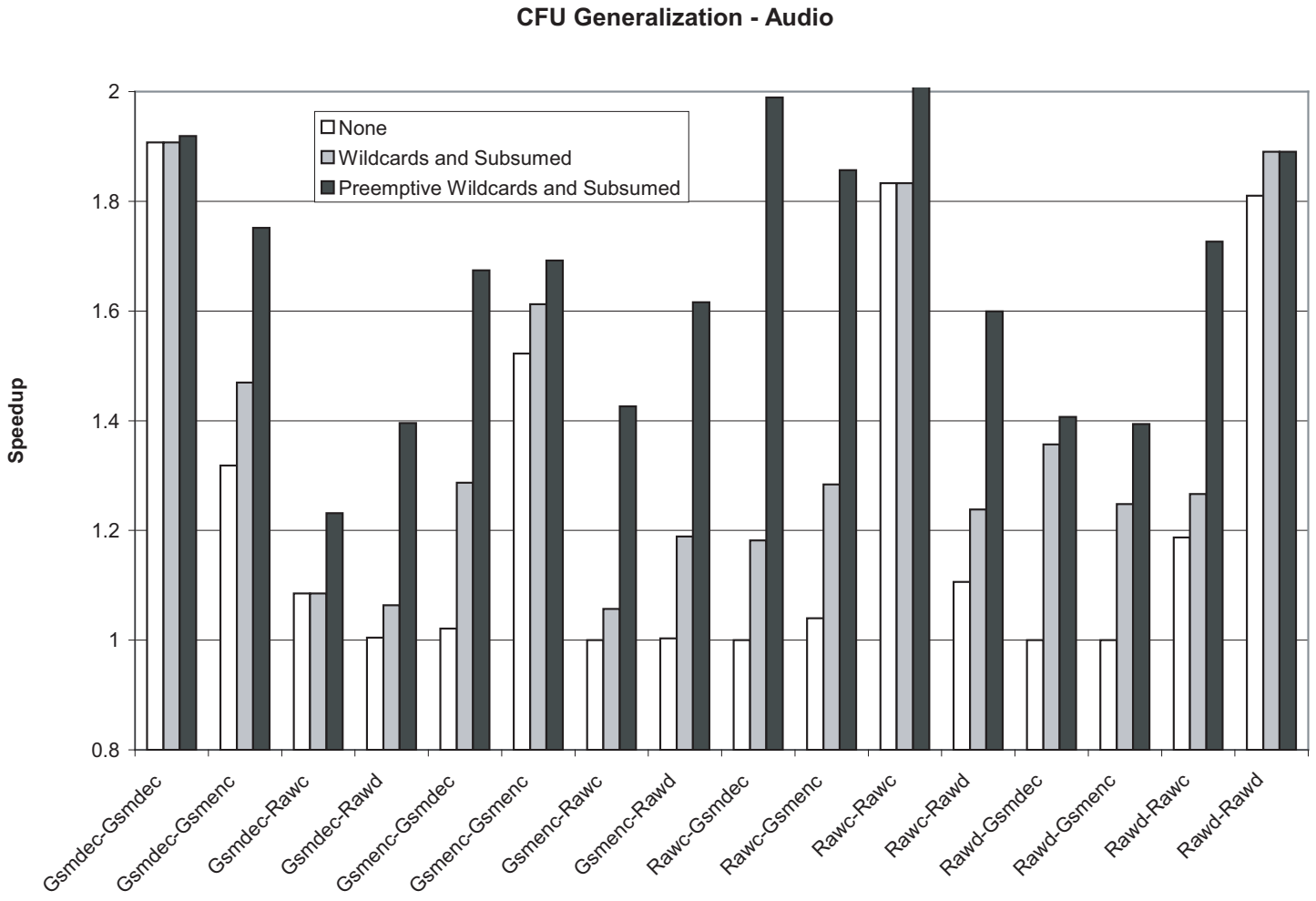


Figure 2.11: Effect of subsumed subgraphs and wildcards in Audio at the 15-adder cost point.

run, and the second one is the application the hardware was designed for. For example, the second set of bars from the left in Figure 2.10, 3des-Blowfish, shows the speedup obtained when running 3des on CFUs designed for blowfish. Each bar in these figures uses the CFUs designed at a cost point of 15 adders. The white bars use CFUs that have no generalization, the gray bars utilize wildcarding and subsumed subgraphs, and the black bars have preemptive wildcarding in addition to the subsumed subgraphs. Although only the encryption and audio domains are shown here, the trends in these two figures hold in both the network and image domains.

One interesting pattern in these figures is that when one application does well using another application's CFUs, it does not necessarily mean that the opposite is true. For example, rijndael does well on rc4's CFUs, but rc4 gets almost no speedup from using rijndael's CFUs without generalization.

The most dominant trend in these figures is that without generalization techniques (i.e. the white bars), most applications do quite poorly when using hardware designed for another application. Rijndael was able to achieve a 1.85 speedup using CFUs designed for rc4, but apart from that, none of the other performance improvements even begin to approach what was achieved with hardware designed specifically for that application. This result was surprising, since applications in the same domain generally have similar DFG structure. The reason this happens is because while the DFGs are similar, they do not match exactly. This serves as strong motivation for the use of CFU generalization techniques for domain-specific acceleration.

As CFUs are generalized (moving to the gray and then to the black bars), it becomes obvious that the critical issue to exploiting CFUs across multiple applications is the ability to map multiple subgraphs onto the CFU hardware. Using opcode classes and subsumed subgraphs allows several applications to approach the speedups attained with CFUs designed specifically for them, e.g. rijndael on rc4, 3des on rijndael, and gsmencode on gsmdecode. Most cross compiles show significant speedups when using generalization techniques, which points toward the conclusion that applications within a domain generally have similar DFG structure in the computationally intense portions of their DFGs.

An important point in Figures 2.10 and 2.11 is that the generalization techniques are

typically not very useful for native compiles. For example, gsmdecode shows little improvement from generalization in Figure 2.11 on CFUs designed for itself. This is because the CFUs are chosen specifically to handle the most computationally intensive portions of the code, leaving few nodes in important parts of the code available to be utilized by wildcard or subsumed subgraphs. The fact that generalization does not help native compiles provides further evidence that the DFG exploration tool does a good job at finding and selecting appropriate CFUs.

Designing CFUs for multiple applications: An alternative strategy to preemptively generalizing CFUs from one application is to design them with multiple applications in mind. This allows for more certainty that the CFUs designed will work across a domain.

Figure 2.12 shows the results of designing CFUs for certain subsets of the encryption domain. The horizontal axis shows which applications the CFUs were designed to target. For example, the middle set of six bars in the left graph shows the speedups of six applications when using CFUs designed for Blowfish, Rc4, and Sha simultaneously, at a cost point of 15 adders. Moving from left to right along the horizontal axis in each graph effectively generalizes the CFUs for the encryption domain, since an additional application is used as input at each step. The left and right graphs show two different paths for generalizing the CFUs.

There are three important trends to note in Figure 2.12. First, adding applications generally improves average performance. For example, moving from the first set of bars to the second set in the left graph, 3des, blowfish, sha, and md5 all improve performance. Rc4 loses a little performance because in the first set, the CFUs were designed specifically for that application and in the second set, part of the area budget is devoted to sha as well. Regardless, the average speedup of the six applications monotonically increases as more applications are taken into account when designing the custom instructions. This is true on the right graph as well.

Second, speedups achieved from the rightmost (domain-wide) set of bars are close to speedups achieved by designing specifically for that application. For example, the speedup for 3des on CFUs designed specifically for it is 2.39 compared to 2.32 on the domain-wide CFUs (rightmost set of bars), and blowfish has a speedup of 1.59 in both the application

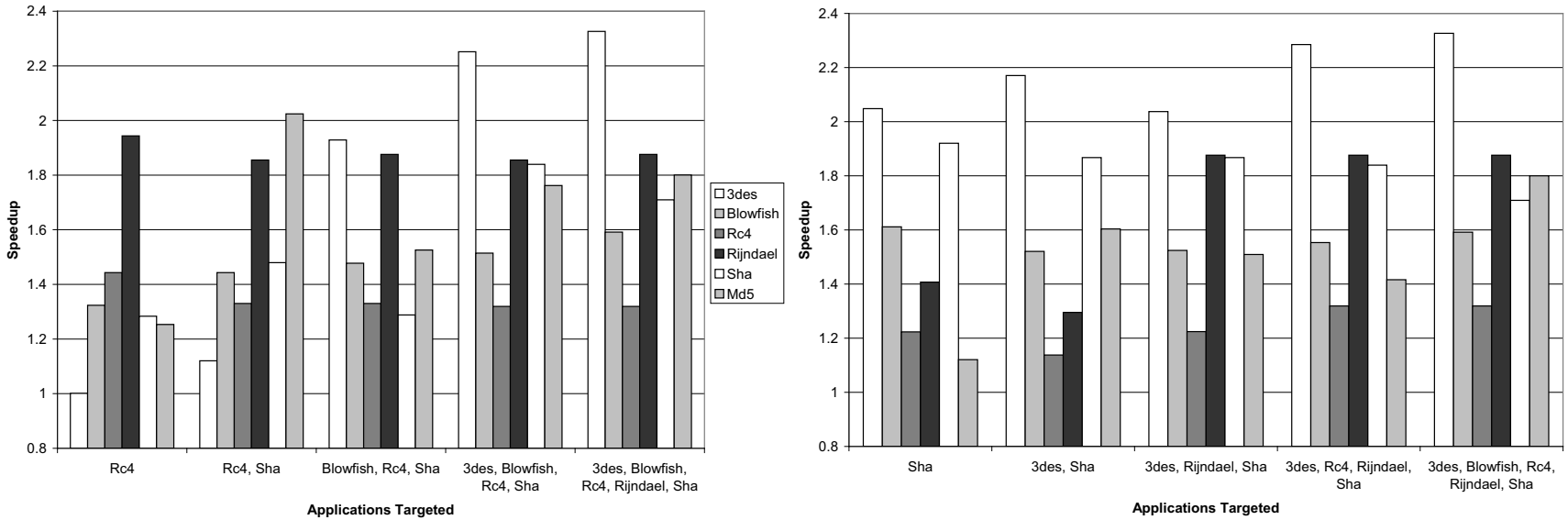


Figure 2.12: Effect of targeting multiple applications

specific and domain-wide CFUs are used. The reason that these speedups are attainable in the domain-wide CFUs is that the DFGs of these applications are quite similar and the generalization techniques that we have proposed allow for creating hardware that maps to the core computational needs of each application.

The last important trend to note in Figure 2.12 is that the md5 application generally improves speedup as the CFUs become more general. Md5 is a program for computing checksums to detect data transmission errors, and is similar in structure to encryption algorithms. Since md5 shows good speedups on the domain-wide CFUs, and they were not designed with this application in mind, it seems likely that these CFUs will be effective on next generation encryption applications.

2.6 Summary

Application-specific instruction set extensions are an efficient way to meet the growing performance and power demands of embedded applications. Designing these extensions has traditionally been very user intensive, as an architect must determine what would make a good extension and manually insert intrinsics into the code to make use of these extensions. In this chapter, we have presented a system that automates this process. Using an efficient dataflow graph exploration heuristic, we are able to discover and automatically select custom function units to meet the demands of an application. We have also demonstrated how a compiler can make use of these custom function units in any application and how to increase their utility through simple generalization techniques.

Our system has demonstrated significant speedups for several applications, with as much as 2.39 for 3des and an average of 1.69, while utilizing modest additional die area. We have shown that typically exact subgraph matches do not occur across applications in a domain, but by using simple generalization techniques (wildcards and subsumed subgraphs) cross-application utilization can be substantially improved. Additionally, we have shown that designing custom instructions with several applications in mind at one time is an effective way to achieve the goals of generalization, and to design with future algorithms in mind.

CHAPTER 3

Generalized Acyclic Accelerators

3.1 Introduction

As covered in the previous chapter, instruction set customization is one method for efficiently providing enhanced performance in processors. By creating application-specific extensions to an instruction set, the critical portions of an application's dataflow graph (DFG) can be accelerated by mapping them to specialized hardware. Though not as effective as ASICs, instruction set extensions improve performance and reduce energy consumption of processors. Instruction set extensions also maintain a degree of system programmability, which enables them to be utilized with more flexibility. An additional benefit is that automation techniques, such as the ones used by ARM OptimoDE, Tensilica, and ARC, have been developed to allow the use of instruction set extensions without undue burden on hardware and software designers.

The main problem with application specific instruction set extensions is that there are significant non-recurring engineering costs associated with implementing them. The addition of instruction set extensions to a baseline processor brings along with it many of the issues associated with designing a brand new processor in the first place. For example, a new set of masks must be created to fabricate the chip, the chip must be reverified (using both functional and timing verification), and the new instructions must fit into a previously established pipeline timing model. Furthermore, extensions designed for one domain are often not useful in another, due to the diversity of computation causing the extensions to

have only limited applicability.

To overcome these problems, the next two chapters focus on strategies to customize the computation capabilities of a processor within the context of a general-purpose instruction set, referred to as *transparent instruction set customization*. The goal is to extract many of the benefits of traditional instruction set customization without having to break open the processor design each time. This is achieved in two steps. First, a compute accelerator is added to the baseline processor design to provide the functionality of a wide range of application-specific instruction set extensions in a single piece of hardware. Next techniques are developed to invoke the accelerators, without augmenting the instruction set.

The focus of this chapter is on design of more general purpose acyclic computation accelerators. An effective design must be capable of executing a wide variety of domain-specific instruction subgraphs faster and more efficiently than a conventional processor pipeline. It must also be both cost effective and power efficient to make its use feasible in an embedded computing environments, and be amenable to efficient run-time control generation. The final issue is the most difficult to quantify, but implies a programmable substrate that is configured with a modest number of control signals.

Two families of designs are presented in this chapter. First, a configurable compute accelerator, or CCA, consists of an array of combinational function units that can efficiently implement many common dataflow subgraphs. The second is a parameterized lookup table (LUT) based accelerator. LUTs are the basis of FPGAs, and natively support any bit-wise function. This power comes at the cost of efficiency, so the LUT-based accelerator presented is tailored to more efficiently target important subgraphs.

A detailed analysis of the CCA and PCFU designs show that they implement the most common subgraphs while keeping control cost, delay, and area overhead to a minimum.

3.2 Related Work

Utilizing instruction set extensions to improve the computational efficiency of applications is a well studied field. Domain specific instruction set extensions have been used in industry for many years, for example Intel's SSE or AMD's 3DNow! multimedia instruc-

tions. Techniques for generating domain specific extensions are typically ad-hoc, where an architect examines a family of target applications and determines what is appropriate.

In contrast to domain specific extensions, a great deal of work has been done on the design of a reconfigurable computation accelerators. Examples include PRISM [8], PRISC [109], OneChip [23], DISC [125], GARP [54], and Chimaera [128]. All of these designs are based on a tightly integrated FPGA, which allows for very flexible computations. However, there are several drawbacks to using FPGAs. One problem is that the flexibility of FPGAs comes at the cost of long latency. While some work [92] has addressed the issue, implementing functions in FPGAs remains inefficient when compared to ASICs that perform the same function. Second, FPGA reconfiguration time can be slow and the amount of memory to store the control bits can be large. To overcome the computational inefficiency and configuration latency, the focus of most prior work dealing with configurable computation units was on very large subgraphs, which allows the amortization of these costs. This work differs in that we focus on acceleration at a finer granularity.

Recent research [130] has proposed using a finer granularity compute accelerator based on slightly specialized FPGA-like elements. By restricting the interconnect of the FPGA-like elements, they reduce the delay of a accelerator without radically affecting the number of subgraphs that can be mapped onto it. While the flexibility to map many subgraphs onto configurable hardware is appealing, there are still the drawbacks of a large number of control bits and the substantial delay of FPGA-like elements.

A key observation we have made is that when collapsing dataflow subgraphs for customized instruction set extensions, the flexibility of an FPGA is generally more than is necessary. FPGAs are designed to handle random computation. The computation in applications is structured using a relatively small number of computational primitives (e.g. add, subtract, shift). Thus, the types of computation performed by instruction set extensions can be implemented much more efficiently by designing a dedicated circuit corresponding to primitives from dataflow graphs. Constructing a circuit of dataflow graph primitives has the additional benefit of keeping the configuration overhead to a bare minimum. This is because selecting from a few primitives is far simpler than selecting from all possible computations. By sacrificing some generality, we are able to achieve a much simpler archi-

	Encryption			MediaBench				SPECInt				
Depth	crc	blowfish	rijndael	djpeg	g721enc	gsmenc	unepic	gzip	vpr	parser	vortex	Average
2	11.13	10.37	4.17	29.79	42.51	41.57	74.87	39.19	44.37	50.39	38.07	47.53
3	11.27	72.29	77.75	38.91	69.38	41.57	95.23	53.48	46.07	82.20	63.49	72.30
4	22.37	81.42	77.75	100	69.38	41.57	100	62.21	95.49	82.54	100	82.61
5	22.37	99.98	100	100	84.71	45.84	100	73.40	99.99	82.54	100	88.85
6	100	100	100	100	84.71	48.77	100	95.46	100	100	100	95.53
7	100	100	100	100	87.24	100	100	100	100	100	100	99.47
$\geq$8	100	100	100	100	100	100	100	100	100	100	100	100

Table 3.1: Cumulative percentage of dynamic subgraphs with varying depths

tecture that still captures the majority of subgraphs.

REMARC [93] and MorphoSys [86] are two designs that also proposed computation architectures more suited for computation of DFG primitives than an FPGA. These coprocessors were geared toward large blocks in multimedia applications, as compared to our design, which executes smaller blocks of computation. Both REMARC and MorphoSys must be programmed by hand to be effectively utilized, since they target large blocks of very regular computation.

Other work [17, 62, 102, 111, 113] proposed subgraph execution structures specifically optimized for linear chains of execution. That is to say these structures only execute subgraphs that have two inputs, one output, and a small number of intermediate nodes. Constraining the subgraphs in this way has been shown to effectively increase the bandwidth of execution resources; however, it restricts the performance increase from dataflow graph compaction [131]. In this work, we develop a more generic architecture, to support the execution of more arbitrary acyclic dataflow subgraphs. These subgraphs are larger than simple linear subgraphs, and attack the computation limitations of processors more than the resource limitations.

3.3 Design of a Configurable Compute Accelerator

The main goal of a CCA is to execute many varied dataflow subgraphs as quickly as possible. A matrix of function units (FUs) is a natural way of arranging a CCA, since it allows for both the exploitation of parallelism in the subgraph and also for the sequential

propagation of data between FUs. In order to be effective, the FUs need to have adequate functionality to support the types of operations that are frequently mapped onto them.

A set of experiments were performed to determine the depth (number of rows) and the width (number of columns) of the matrix of FUs, as well as the capabilities of each FU. Using the SimpleScalar toolset [9] for the ARM instruction set, traces were collected for a set of 29 applications. The application set consisted of four encryption related algorithms and selected MediaBench and SPECint benchmarks. The goal of this benchmark set was to represent a wide variety of integer computational behavior.

Traces from these benchmarks were analyzed offline using the optimal discovery algorithm (described in section 4.3.2) to determine the important subgraphs a CCA should support. The characteristics of these subgraphs were then used in determining the configuration of our proposed CCA. The subgraphs were weighted based on execution frequency to ensure that heavily utilized subgraphs influenced the statistics more. Because dynamic traces are used as the basis for analysis, conservative estimates have to be made with regards to which operation results must be written to the register file. That is, unless a register is overwritten within the trace, it must be written to the register file, because it may be used elsewhere in the program. This potentially restricts the size of subgraphs available to offline replacement schemes, however it accurately reflects what is necessary for supporting runtime replacement techniques.

The subgraphs considered in this study were limited to have at most four inputs and two outputs. Further, memory, branch, and complex arithmetic operations were excluded from the subgraphs as will be discussed later in the section. Previous work [131] has shown that allowing more than four input or two output operands results in very modest performance gains when memory operations are not allowed in subgraphs, thus the input/output restriction is considered reasonable.

A similar characterization of subgraphs within traces was performed previously [117]. This differs from the analysis here in that we gear experiments specifically toward the design of a CCA. The previous work proposed many additional uses for frequently occurring subgraphs, for example cache compression and more efficient instruction dispersal.

	1	2	3	4	5	6	7
1	100.00	59.02	22.89	13.14	6.48	4.20	0.25
2	91.11	50.57	9.93	4.10	0.59	0.15	0.01
3	57.39	17.79	6.25	2.89	0.09	0.02	0.01
4	18.53	8.27	1.58	0.11	0.02	0.01	0.00
5	8.65	2.06	0.14	0.04	0.01	0.01	0.00
6	2.13	1.23	0.09	0.01	0.01	0.00	0.00
7	1.23	0.10	0.07	0.01	0.00	0.00	0.00
8	0.11	0.07	0.01	0.00	0.00	0.00	0.00

Table 3.2: Matrix utilization of subgraphs

3.3.1 Analysis of Applications

The matrix of FUs comprising a CCA can be characterized by the depth, width, and operation capabilities. Depth is the maximum length dependence chain that a CCA will support. This corresponds to the potential vertical compression of a dataflow subgraph. Width is the number of FUs that are allowed to go in parallel. This represents the maximum instruction-level parallelism (ILP) available to a subgraph. The operation capabilities are simply which operations are permitted in each cell of the matrix.

Depth of Subgraphs: Table 3.1 shows the percentage of subgraphs with varying depths across a representative subset of the three groups of benchmarks. For example, the 81.42% in blowfish at depth 4 means that 81.42% of dynamic subgraphs in blowfish had depth less than or equal to 4. Although only 11 benchmarks are displayed in this table, the final column displays the average of all 29 applications run through the system. On average about 99.47% of the dynamic subgraphs have depth 7 or less. Since the depth of the CCA directly affects the latency through it, depth becomes a critical design parameter. It can be seen that a CCA with depth 4 can be used to implement more than 82% of the subgraphs in this diverse group of applications. Going below depth of 4 seriously affects the coverage of subgraphs implementable by the CCA. Therefore, only CCAs with maximum depth of 4 to 7 are considered.

Width of Subgraphs: Table 3.2 shows the average width statistics of the subgraphs for the 29 applications. A value in the table indicates the percentage of dynamic subgraphs that had an operation in that cell of the matrix layout (higher utilized cells have a darker

Uop	Opcode Semantics	Percentage
ADD	addition	28.69
AND	logical AND	12.51
CMP	comparison	0.38
LSL	logical left shift	9.81
LSR	logical right shift	2.37
MOV	move	11.66
OR	logical OR	8.66
SEXT	sign extension	10.38
SUB	subtract	4.82
XOR	logical exclusive OR	5.09

Table 3.3: Mix of operations in common subgraphs

background). For example, 4.2% of dynamic subgraphs had width 6 or more in row 1. Only 0.25% of subgraphs had width 7 of more, though. Similar cutoffs can be seen in the other rows of the matrix, such as between widths 4 and 5 in row 2. This data suggests that a CCA should be triangular shaped to maximize the number of subgraphs supported while not needlessly wasting resources.

FU Capabilities: Table 3.3 shows the percentage of various operations present in the frequent subgraphs discovered in above set of benchmarks. Operations involving more expensive multiplier/divider circuits were not allowed in subgraphs, because of latency considerations. Additionally, memory operations were also disallowed. Load operations have non-uniform latencies, due to cache effects, and so supporting them would entail incorporating stall circuitry into the CCA. This would increase the delay of the CCA and make integration into the processor more difficult.

Table 3.3 shows that 48.3% of operations involve only wires (e.g. SEXT and MOV) or a single level of logic (e.g. AND and OR). Another 33.9% of operations (ADD, CMP, and SUB) can be handled by an adder/subtracter. Thus, the adder and the wire/logic units were the main categories of FUs considered for the design of a CCA. Although shifts did constitute a significant portion of the operation mix, barrel shifters were too large and incurred too much delay for a viable CCA.

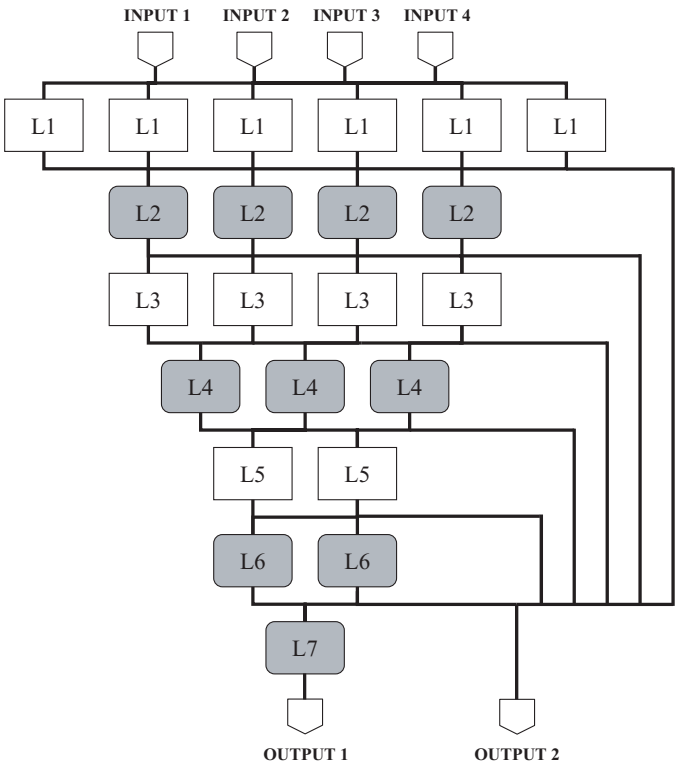


Figure 3.1: Block diagram of the depth 7 CCA

3.3.2 Proposed CCA Design

The proposed CCA is implemented as a matrix of heterogeneous FUs. There are two types of FUs in this design, referred to as type A and B for simplicity. Type A FUs perform 32-bit addition/subtraction as well as logical operations. Type B FUs perform only the logical operations, which include and/or/xor/not, sign extension, bit extraction, and moves. To ease the mapping of subgraphs onto the CCA, each row is composed of either type A FUs or type B FUs.

Figure 3.1 shows the block diagram of a CCA with depth 7. In this figure, type A FUs are represented with white squares and type B FUs with gray squares. The CCA has 4 inputs and 2 outputs. Any of 4 inputs can drive the FUs in the first level. The first output delivers the result from the bottom FU in the CCA, and the second output is optionally driven from an intermediate result from one of the other FUs.

The outputs of the FUs are fully connected to the inputs of the FUs in the subsequent

Depth	Configuration	Control	Delay	Cell area	FPGA delay
4	6A-4B-3A-2B	172 bits	3.19 ns	$0.38\ mm^2$	18.84 ns
5	6A-4B-4A-2B-1B	197 bits	3.50 ns	$0.40\ mm^2$	19.97 ns
6	6A-4B-4A-3B-2A-1B	229 bits	4.56 ns	$0.45\ mm^2$	24.86 ns
7	6A-4B-4A-3B-2A-2B-1B	245 bits	5.62 ns	$0.48\ mm^2$	25.39 ns

Table 3.4: CCA configurations and synthesis results

row. The decision to only allow units to talk to the next row was made to keep the amount of control to a minimum. As the outputs of one row and the inputs of the next are fully connected, the interconnect network is expensive in terms of delay. This delay was necessary, however, to reduce the complexity of the dynamic discovery and selection algorithms described in the next section.

The critical path of adder/subtracter circuits is much longer than any of the other operations supported by the CCA. To control the overall delay, the number of rows with adders is restricted. More than 99.7% of dynamic subgraphs can be executed on a CCA with 3 adders in serial, and so the depth 7 CCA in Figure 3.1 is restricted to having 3 rows of type A FUs. Further, restricting the CCA to only 2 rows of type A FUs allows it to support only 91.3% of the subgraphs, but significantly improving the delay of the CCA. The type A and type B rows were interspersed within the CCA, because empirical analysis shows many of the subgraphs perform a few logic operations between subsequent additions. This is particularly true in the encryption applications.

Four CCA models were synthesized using Synopsys CAD tools with a popular standard cell library in 0.13μ technology. Each model has different depth and row configurations, shown in Table 3.4. The configurations in this table indicate the number and type of FUs in each row, from top to bottom. For example, the depth 4 CCA has 6 type A FUs in row 1 and 4 type B FUs in row 2. Delay of the CCA and the die area are also listed in this table. The depth 4 CCA had a latency of 3.19ns from input to output and occupied $0.38mm^2$ of die area. The last column of Table 3.4 contains the delay of each CCA design when synthesized on an FPGA[1]. It suggests that FPGAs may not be a suitable device for

[1]Xilinx Virtex-II Pro family, based on 0.13μ technology

an efficient implementation of the CCA at this granularity of subgraph, though we did not perform any measurements of direct realization of the applications' subgraphs via the FPGA.

The control bits needed for each model are also shown in Table 3.4. Each FU has four opcode bits that define its functionality. Since the output of each FU is connected to every input port of the FUs in the next level, signals to control the bus are required. The number of those signals corresponds to twice the number of FUs in the next level, considering there are two input ports for each FU and each output could feed each input. Control bits for which FU provides the second output are also needed. The total number of control bits was a critical factor in the design of these CCAs.

3.3.3 Integrating the CCA into a Processor

In the context of a processor, the CCA is essentially just another FU, making integration into the pipeline fairly straightforward. The only datapath overhead consists of additional steering logic from reservation stations and bypass paths from the CCA outputs. The CCA itself is not pipelined, removing the complexity of having to introduce latches in the matrix of FUs or having to forward intermediate results from internal portions of the matrix.

Accommodating a 4 input, 2 output instruction into the pipeline is slightly more complicated. One potential way to accomplish this is to split every CCA operation into 2 uops, each having 2 inputs and 1 output. By steering the 2 uops consecutively to a single CCA, a 4 input, 2 output instruction can be constructed without altering register renaming, the reservation stations, the re-order buffer, or the register read stage. The downside to this approach is that the scheduling logic is complicated by having to guide the two uops to the same CCA.

Interrupts are another issue that must be considered during CCA integration. The proposed CCA was intentionally designed using simple FUs that cannot cause interrupts. However, splitting the CCA operation into 2 uops means that an external interrupt could cause only half of the operation to be committed. To avoid this problem, the 2 uops must be committed atomically.

Pipeline	4 wide
RUU size	128
Fetch Queue Size	128
Execution Units	4 simple ALUs, 2 multipliers, 2 memory ports
Branch Predictor	12-bit gshare
Frame Cache	32k uops, 256 inst traces
L1 I-cache	32k, 2 way, 2 cycle hit
L1 D-cache	32k, 4 way, 2 cycle hit
Unified L2	1M, 8 way, 12 cycle hit
Memory	100 cycle hit
Frame Cache Discovery and Replacement Latency	5000 cycles

Table 3.5: Processor configuration

Control bits for the CCA can be carried along with the 2 uops. Since there is at most 245 bits of control necessary in the proposed CCAs, this means that each uop would carry around 130 bits, which is roughly the size of a uop in the Intel P6 microarchitecture.

3.4 Experimental Evaluation

The proposed accelerators were modeled in the SimpleScalar simulator [9] using the ARM instruction set. Within SimpleScalar, some ARM instructions are broken into micro-operations, e.g., load multiple, which performs several loads to a continuous sequence of addresses. Many ARM instructions allow for an optional shift of one operand, and it is important to note that these shifts are also broken into uops. Since our CCA does not support shifts, it would otherwise not be possible to execute these operations on the CCA.

The simulated processor model is a 4-issue superscalar with 32k instruction and data caches. More details of the configuration are shown in Table 3.4. Consistent with Section 3.3, the benchmarks used in this study consist of 29 applications from SPECint, MediaBench, and four encryption algorithms. We select a representative subset of the applications to show in our graphs, consisting of four SPECint applications (175.vpr, 181.mcf, 186.crafty, and 255.vortex), six MediaBench applications (djpeg, cjpeg, epic, mpeg2enc,

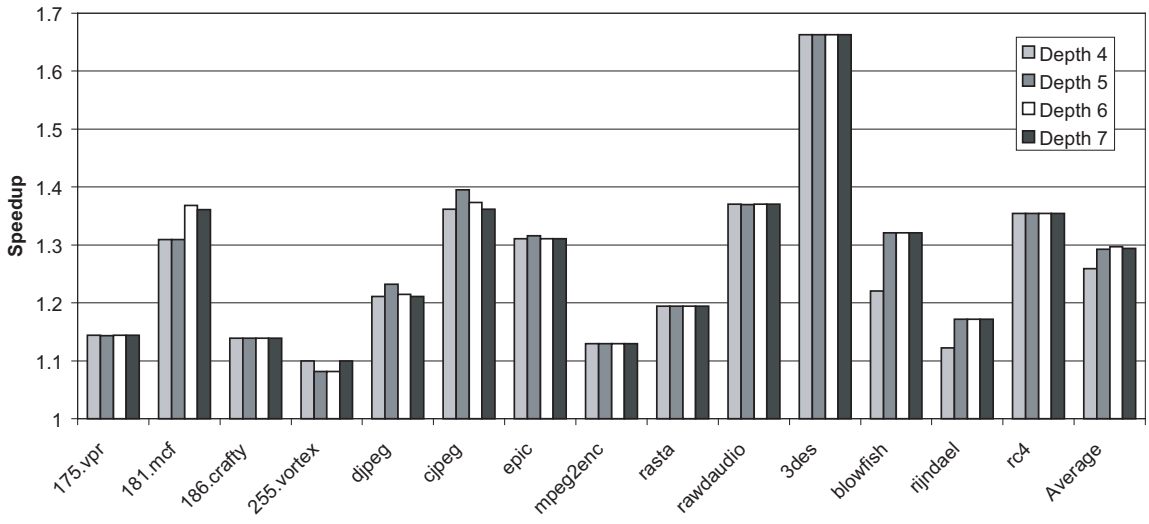


Figure 3.2: Varying the CCA configurations

rasta, and rawdaudio) and four popular encryption applications (3des, blowfish, rijndael and rc4). Each benchmark was run for 200 million instructions, or until completion. The initial 50 million instructions of each SPEC benchmark were skipped to allow the initialization phase of the benchmark to complete. All of the benchmarks were compiled using gcc with full optimizations.

Figure 3.2 compares the performance across varying depth CCAs using an offline discovery algorithm and retirement-based replacement (described in the next chapter). Speedups are calculated as the ratio of execution cycles without and with the CCA of the specified configuration. The configuration of the CCAs match the descriptions in Table 4.1 and all have a latency of one. From the graph, the most obvious result is the flatness of each set of bars. Little performance is gained as larger CCA designs are utilized. However, this result could be anticipated as it agrees with the depth statistics observed in Table 3.1. Generally, adding depth to the subgraph beyond 4 provides only modest gains in coverage (depth 4 covers 82%). Further, the large, important subgraphs that would have been executed on the 7-deep CCA can simply be broken into two subgraphs executed on the smaller CCAs. As long as there is enough ILP and the large subgraph is not on the critical path, this additional reduction of latency achieved with a larger CCA will not significantly improve

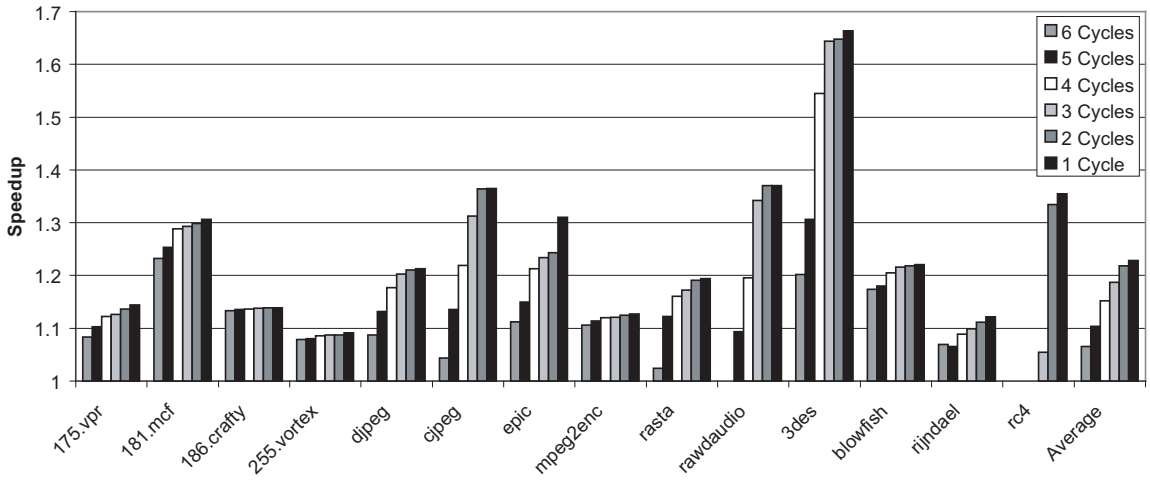


Figure 3.3: The effect of CCA latency on speedups

performance.

There are a number of notable exceptions to the flat behavior. For example, some benchmarks show a performance jump at one particular CCA size. For instance, blowfish from depth 4 to depth 5. This is because a critical subgraph was admitted at that point. Interestingly, sometimes adding depth actually hurts the performance, as in the case of cjpeg. This is because of second order effects involved with subgraph discovery. Sometimes creating a CCA operation out of a large 7-deep subgraph, while optimal from the coverage standpoint, is not as effective as creating two smaller subgraphs.

Figure 3.3 shows the affect of CCA latency on overall performance. This graph reflects static discovery, retirement-based replacement and a CCA of depth 4. Speedup is calculated in the same manner as in the previous graph. This figure shows that the effect of CCA latency is highly dependent on the application. For example, rc4's speedup rapidly declines when the latency is increased, reaching zero for latency 3 and beyond. This is because rc4 has one dominant critical path on which all the subgraphs appear. Since the subgraphs are all on the critical path, the performance is highly sensitive to the number of cycles to execute each one.

On the other hand, 186.crafty suffers little penalty from the added latency of the CCA. This behavior is generally attributed to one of two reasons. First, the critical path is memory

bound, thus CCA latency is a second order effect. Second, the application has enough ILP so that longer CCA latencies are effectively hidden. Such applications benefit from more efficient execution provided by the CCA, but are less sensitive to latency. Other applications, such as 3des and rawdaudio, degrade slightly at small latencies (e.g., 1-3 cycles), then fall off sharply at larger latencies (e.g., 4 or 5 cycles). This reflects the point at which the CCA instructions become the critical path because of their added latency. As the latency increases, benefits from vertically compressing the dataflow graph disappear. The speedups that remain are solely due to the additional parallelism provided by the CCA.

3.5 CCA Summary

So far in this chapter, we have presented a novel mechanism to accelerate application performance through the use of a configurable compute accelerator, or CCA. A CCA is a group of function units connected in a matrix-like configuration, added to a general-purpose core to implement dataflow subgraphs. Subgraphs from a stream of processor instructions are identified and mapped onto this CCA.

Our experiments reveal that significant speedups are possible for a variety of applications, both from the embedded and general-purpose computing domains. The speedup was up to 66% for a 4-deep CCA (26% on average), and the area overhead is reasonably small. The CCA has a moderate degree of latency tolerance, and thus can be more easily integrated into any modern processor pipeline.

3.6 The Programmable Carry Function Unit

The remainder of this chapter presents the programmable carry function unit, or PCFU, an alternate design for acyclic computation accelerators. The PCFU is a lookup table (LUT) based accelerator. LUT based accelerators were previously introduced in [130], where every bit was given a separate LUT configuration. Here, a generalization of the prior technique is used. The PCFU leverages the design of carry look-ahead adders to break the cascaded tree of LUTs in the original design, creating a faster and more efficient design.

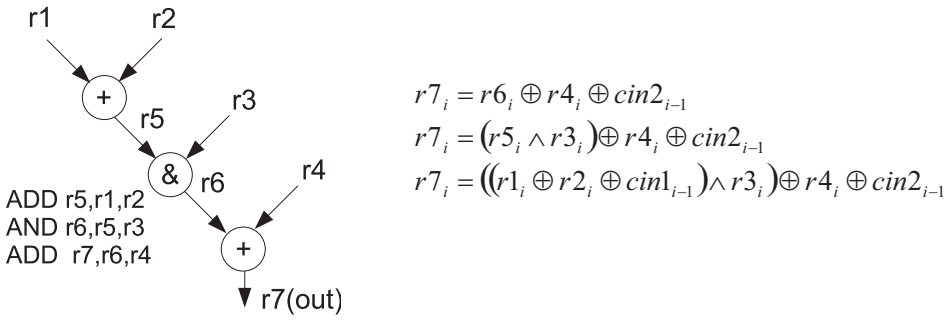


Figure 3.4: An example dataflow subgraph and the output expressed as a function of the inputs.

The PCFU is not a single design, but rather specifies a parameterized design space that offers complex tradeoffs between subgraph execution capabilities (programmability) with the cost and worst-case delay (efficiency) of the substrate. Thus, we will now present a systematic exploration of the PCFU design space. We examine the critical tradeoffs associated with designing LUT based arrays including LUT size, number of carry signals that are propagated, and support for non-LUT operators, such as shift. To perform this exploration, a complete compilation and simulation system for PCFUs based on the ARM-9 processor are used. PCFU designs are developed in Verilog and synthesized to measure area and delay.

3.7 PCFU Operation and Design Space

The design of a generalized accelerator substrate on which dataflow subgraphs are executed is a major challenge. The accelerator should be programmable enough to cover most of the recurrent subgraphs, and at the same time be easy to configure and have low latency to execute subgraphs efficiently.

In this section, we describe a LUT-based accelerator, the Programmable Carry Function Unit (PCFU). The PCFU can execute subgraphs of any number of logical operations and a predefined number of additions/subtractions. The PCFU offers the advantage of being sufficiently programmable to cover a wide variety of subgraphs, while maintaining a relative

low interconnect complexity and latency compared to FPGA devices.

3.7.1 Principles

The PCFU approach builds on the principles introduced in [130], with the basic idea being to extract a logical expression of each output bit as a function of the inputs to the subgraph. Given this logical expression, a LUT stores the truth table corresponding to this expression, which is used later to directly compute the output given the inputs.

Consider the example of Figure 3.4. Each bit of the output register, `r7`, can be expressed as a logical function of `r1`, `r2`, `r3`, `r4` and the carry bits from any additions in the subgraph. For example, the output function for bit `i` of the output of the subgraph can be expressed as $fr7_i = r6_i \oplus r4_i \oplus cin2_{i-1}$, since this is the definition of a bit-wise add. Next, $r6_i$ can be re-expressed as $(r5_i \wedge r3_i)$, yielding the second equation in Figure 3.4. Likewise, $r5_i$ can be expressed as a function of $r1$, $r2$, and the carry signal generated by the first addition in the subgraph. Once this is done, each bit of $r7$ is expressed as a logical function of only the inputs and the carry signals, shown at the bottom of Figure 3.4. Using this process allows for expressing any sequence of logical, integer instructions as a function of the input registers of the subgraph. This enables direct mapping of subgraphs into lookup tables, with the only difficulty being the need to calculate the carry signals.

Figure 3.5 shows the accelerator proposed in [130], called the *Function* unit. In this design, each carry bit is calculated and forwarded to the higher significant bit. The LUTs $fr7_iLUT$, $cin1_iLUT$, and $cin2_iLUT$ implement the functions `fr7`, `cin1`, and `cin2` respectively. The *Function* unit provides fine grain programmability and flexibility by specifying different LUT configurations for each output and carry bit. This high flexibility comes at the cost of high latency because of the ripple scheme to propagate the carry to upper significant bits. Also, this accelerator requires a large amount of control data to configure each bit and their associated carries.

The PCFU is also a LUT-based accelerator, but avoids both the large configuration and the high latency of the ripple carry propagation by defining one LUT (and associated carry LUTs) for all output bits. Aside from saving on-chip space, this approach allows us to

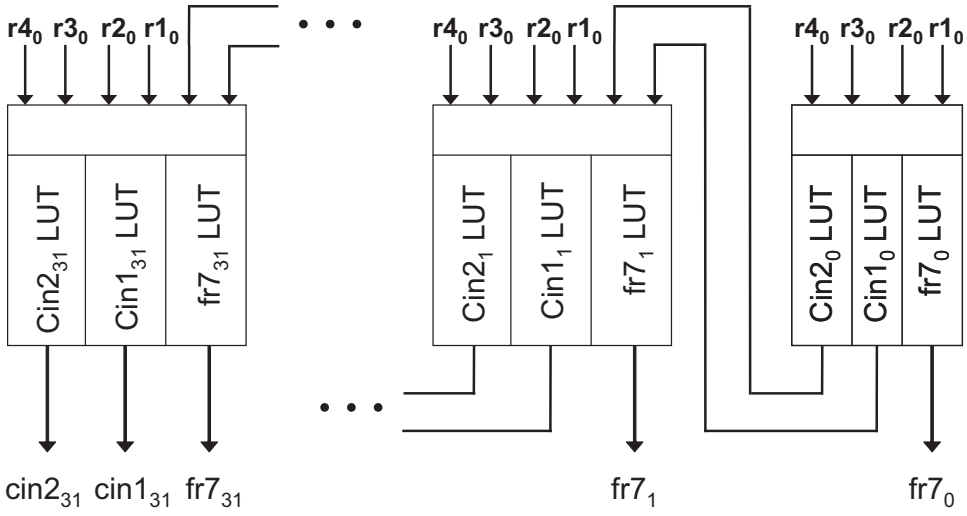


Figure 3.5: Organization of the LUT based *Function* unit from prior work.

leverage fast carry propagation schemes, such as Kogge-Stone [67] or Brent-Kung [18] parallel prefix adders.

Most of the existing fast carry propagation techniques are based on first calculating a (g_i, p_i) pair [68], where given inputs bits a_i and b_i, $g_i = a_i \wedge b_i$ (generate), and $p_i = a_i \oplus b_i$ (propagate). If subtraction is done instead of addition, b_i is replaced by $\bar{b_i}$. This p-g pair of values is then fed to a carry propagation network to calculate the carry bits. The PCFU design generalizes the calculation of the (g_i, p_i) pair by creating a pair of LUT configurations (gi LUT - pi LUT) for each addition/subtraction. For example, in Figure 3.4 the (g,p) pairs of the 2 additions can be expressed as $g1_i = r1_i \wedge r2_i$, $p1_i = r1_i \oplus r2_i$, $g2_i = ((r1_i \oplus r2_i \oplus cin1_{i-1}) \wedge r3_i) \wedge r4_i$, and $p2_i = ((r1_i \oplus r2_i \oplus cin1_{i-1}) \wedge r3_i) \oplus r4_i$. By separately computing the carry signal using these LUTs and carry propagation networks, the PCFU breaks the dependence of output bits on the values of lower order input bits. That is, bit 31 of the output is not a function of bit 0 of the input values as long as we have the carry signal precomputed. This enables the PCFU to have a much lower latency than most FPGA-based accelerator designs, which need to propagate the carry signal from lower order bits.

Figure 3.6 shows the design of a PCFU that can collapse a sequence of dependent

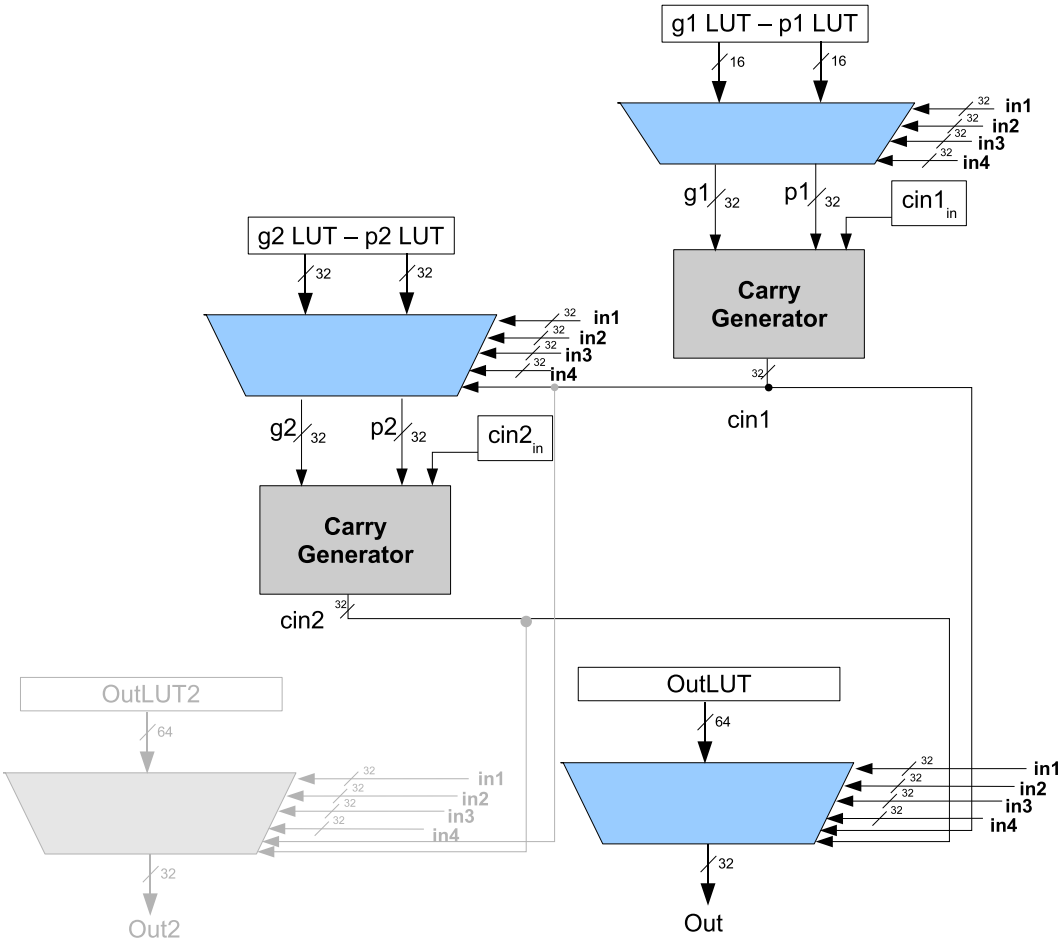


Figure 3.6: Baseline PCFU design.

instructions with up to two additions or subtractions, and *any* number of logical operations, given a fixed number of inputs. Note that, although Figure 3.6 may suggest that the two additions/subtractions need to be dependent, the PCFU can collapse any two addition/subtraction regardless of their position in the subgraph. That is, they may be in parallel, dependent or even interleaved with other logical operations.

For a given subgraph, the basic idea of the PCFU is to generate a LUT (OutLUT) configuration for the output function and appropriate configurations (gi LUT and pi LUT) to generate the carries for each individual addition/subtraction in the subgraph. The purpose of the $cini_{in}$ signal is to implement subtractions. The primary benefit of using the PCFU

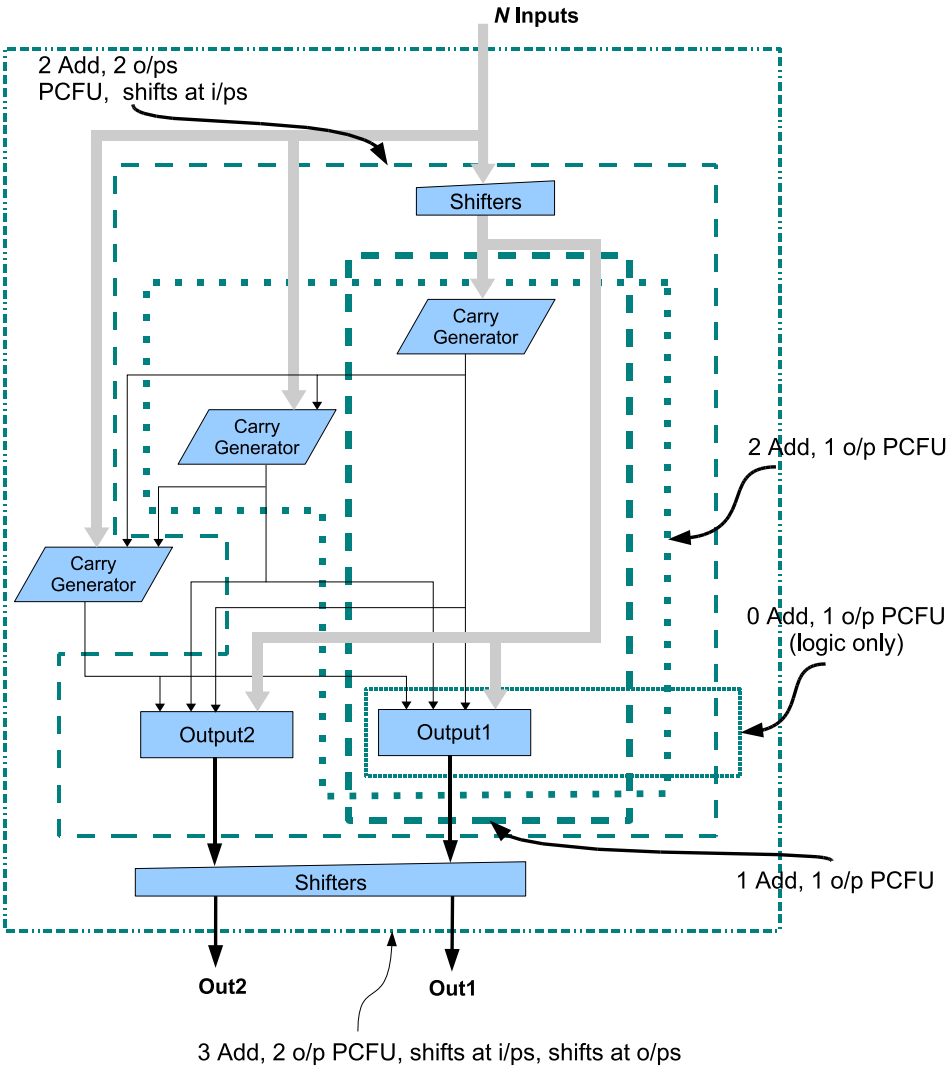


Figure 3.7: PCFU design space.

over previous work [130] is the use of more advanced carry generation networks and fewer configuration bits in the accelerator.

3.7.2 PCFU Design Space

Figure 3.7 shows basic building blocks of a generalized PCFU that can support N inputs, 3 additions/subtractions, 2 outputs, and shift operations at the inputs and outputs of the dataflow subgraph. The basic building blocks of the PCFU are the carry generator for each

addition/subtraction supported and an output LUT for each subgraph output supported.

Increasing the number of outputs supported by the PCFU is a fairly straight forward process, which only requires adding an output LUT in parallel with the already existing output LUTs. None of the other structures in the PCFU are affected.

Supporting additional inputs is more complicated, since it involves increasing the size of the LUTs for the carry generators and the output LUTs. This is because the logical function for each bit depends on another Boolean variable (the new input), which doubles the size of each truth table used to compute results.

Similar to increasing the number of inputs, increasing the number of adds that are supported doubles the size of the output LUTs, since the outputs are now a function of another carry-in signal. Beyond this, supporting more adds requires an additional set of carry propagate LUTs, which are dependent on the inputs and all previous carry-in signals. This means that the added carry-propagate LUT is larger than all the previous carry propagate LUTs combined. A new carry generation tree lies directly on the critical path of the PCFU, as well.

Supporting shift operations within subgraphs is desirable, but infeasible on the PCFU. Allowing shifts would make each output bit a function of every input bit, instead of the small number of input bits in the proposed design. This would make the LUTs very large. However, separate shifters may be added at the inputs and/or the outputs of the PCFU to support shift operations at the inputs and outputs of the dataflow subgraphs. This would not change the size of the LUTs, but would lengthen the critical path of the PCFU.

Each of these vectors in the design space is explored in Section 3.8.

3.8 Exploring the PCFU Design Space

The purpose of this section is to evaluate the different tradeoffs involved in designing a PCFU for subgraph acceleration. The designs are evaluated using latency of the PCFU, die area consumed by the PCFU, as well as performance improvement of the PCFU-augmented processor.

Evaluation of the performance improvement achieved using PCFUs was done using

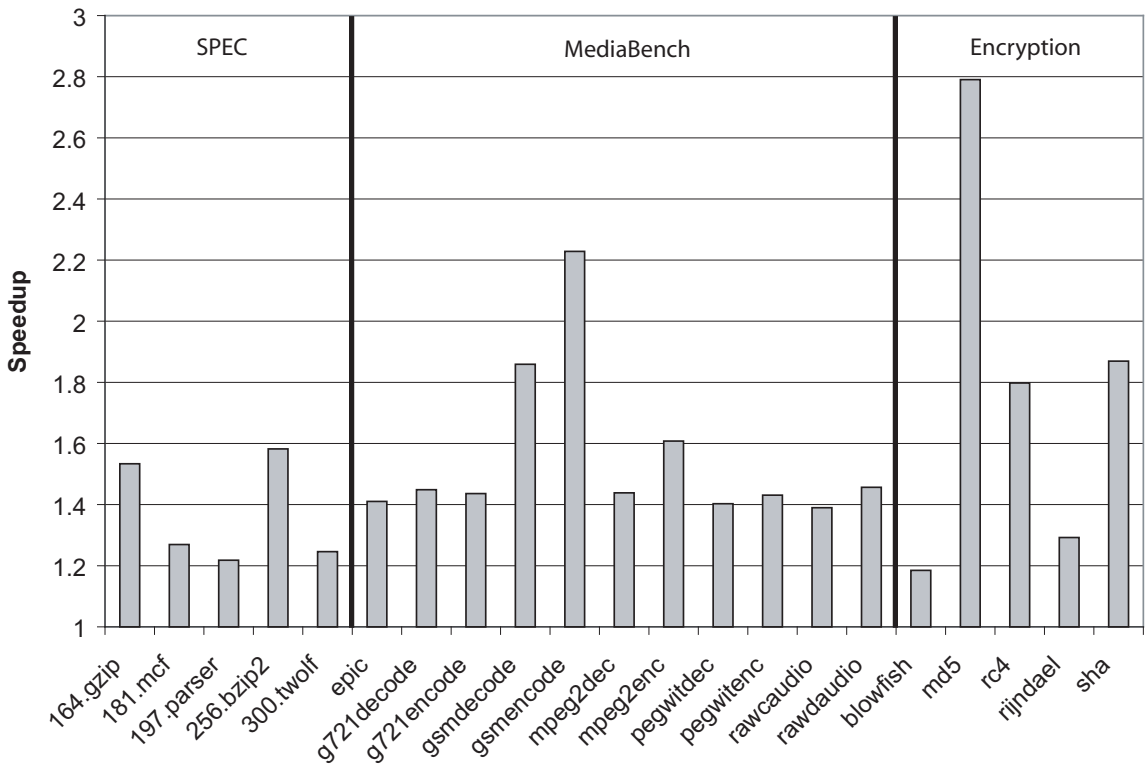


Figure 3.8: Effectiveness of the baseline 4-input, 2-output PCFU design.

a version of the Trimaran compiler [121] ported to the ARM instruction set. The compiler was augmented with a parameterized subgraph matching engine, which allowed us to easily change the types of subgraphs selected based on the characteristics of the underlying hardware. After subgraphs selection, binaries were created using the GNU assembler/linker, and simulated using SimpleScalar ARM [9]. The simulator was configured to model an ARM 926EJ-S processor [5], a popular single-issue embedded core with a five stage pipeline.

In order to determine the latency and area properties of the PCFUs, several designs were synthesized, including place-and-route. The designs were implemented using Synopsys tools with a standard cell library in 0.13μ. The critical path latencies are reported, as well as the die areas given both in mm^2 and as the percentage area of an ARM926EJ-S core without caches. Note that not every design simulated, was synthesized, since creating and

Design	Latency (ns)	Area (mm^2)	Area (% of ARM926EJ-S)
2 In, 1 Out, 2 Adds, No Shift	3.03	0.052	2.3
2 In, 2 Out, 2 Adds, No Shift	2.66	0.056	2.5
3 In, 1 Out, 2 Adds, No Shift	3.24	0.068	3.1
3 In, 2 Out, 2 Adds, No Shift	3.32	0.100	4.5
4 In, 1 Out, 2 Adds, No Shift	3.79	0.134	6.1
4 In, 2 Out, 2 Adds, No Shift	4.20	0.171	7.7
4 In, 3 Out, 2 Adds, No Shift	4.57	0.230	10.4
5 In, 1 Out, 2 Adds, No Shift	5.25	0.214	9.7
5 In, 2 Out, 2 Adds, No Shift	5.30	0.306	13.9
5 In, 3 Out, 2 Adds, No Shift	5.40	0.397	18.0
6 In, 1 Out, 2 Adds, No Shift	5.47	0.465	21.1
6 In, 2 Out, 2 Adds, No Shift	5.27	0.600	27.2
6 In, 3 Out, 2 Adds, No Shift	5.87	0.787	35.8

Table 3.6: Synthesis results for PCFU designs with varying numbers of inputs and outputs.

verifying HDL for the PCFUs is a very time consuming process.

We chose to evaluate our designs using benchmarks from the SPECint2000 and MediaBench [75] benchmark suites, as well as several encryption kernels. Full runs of each benchmark using the training input set were performed. Applications from the two benchmark suites that do not appear were omitted either due to very long runtime (in the case of 254.gap), or limitations in the compiler infrastructure.

Baseline Design. In order to explore the various dimensions of the PCFU design space, we first define a starting point. Previous work [28] has shown that a subgraph execution unit with 4 inputs and 2 outputs, supporting two adds is a reasonable design choice. As such, this is baseline for our evaluation. The design of this PCFU can be seen in Figure 3.6.

The speedups attained using this baseline design are presented in Figure 3.8 for the three groups of benchmarks. Unless otherwise noted, simulation was done assuming that the PCFU requires one cycle to execute the subgraph and does not affect the cycle time of the processor. The main point to take from this figure is the magnitude of the bars. On average, a speedup of 1.62 over the baseline processor is observed, with a maximum of 2.79. This shows that transparent instruction set customization using a PCFU is an effective way to improve the performance of embedded processors. Also note that the

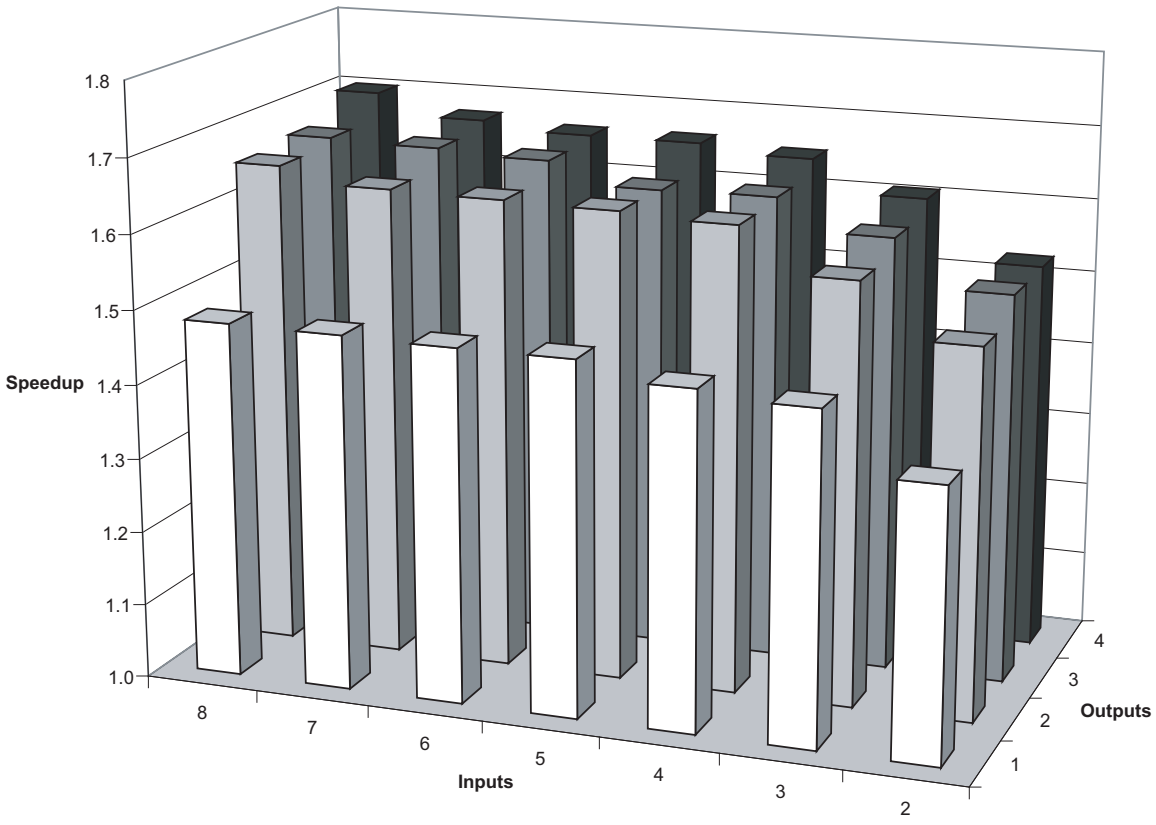


Figure 3.9: Effectiveness of PCFU designs with varying numbers of inputs and outputs.

speedup varies a great deal from application to application. This is correlated to the size of the computation subgraphs available for execution on the PCFU. Since subgraphs are bounded by memory operations, applications that perform a large amount of computation (especially logic operations) between memory accesses benefit the most.

Design Space Parameters. The PCFU design space is evaluated along three independent axes: number of inputs/outputs, number of additions, and support for shift operations. The number of additions specifies the number of carry chains that the PCFU implements. In varying the number of additions, it is also possible to emulate PCFUs with larger number of additions by connecting smaller PCFUs in series, e.g., a 2-adder PCFU can be emulated by connecting two 1-adder PCFUs in series. Shift operations are not supported directly by the PCFU, but by creating hybrid accelerator substrates consisting of PCFUs and shifters.

Number of Inputs/Outputs. The first design space parameter is the effect that the

Design	Latency (ns)	Area (mm^2)	Area (% ARM926EJ-S)
4 In, 2 Out, 0 Adds, No Shifts	0.62	0.042	1.9
4 In, 2 Out, 1 Adds, No Shifts	2.44	0.095	4.3
4 In, 2 Out, 2 Adds, No Shifts	4.20	0.171	7.7
4 In, 2 Out, 3 Adds, No Shifts	5.78	0.361	16.4
4 In, 1 Out, 2 Adds, No Shifts	3.79	0.134	6.1
4 In, 1 Out, 2 (1-1) Adds, No Shifts	3.77	0.116	5.3
4 In, 1 Out, 3 Adds, No Shifts	5.82	0.274	12.4
4 In, 1 Out, 3 (2-1) Adds, No Shifts	6.50	0.212	9.6
4 In, 1 Out, 3 (1-1-1) Adds, No Shifts	6.10	0.180	8.1

Table 3.7: Synthesis results for PCFU designs with varying numbers of additions supported.

number of inputs and outputs has on the system. These parameters are very important, as they have a strong impact on the types of graphs that can be executed on the PCFU. The number of inputs/outputs in the PCFU also has an effect on the register file since each of the inputs/outputs must be read from or written to it. This means that a large number of inputs or outputs requires a larger register file, multiple cycles to read and write results, or "shadow register files" to increase the operand bandwidth without increasing the latency. All of these options carry overheads.

From the perspective of PCFU design, the number of inputs must be carefully controlled. Increasing the number of inputs by one means that each output bit is the function of another binary variable. This essentially doubles the size of each LUT in the design. Aside from the exponential increase in area, this LUT size increase also causes the overall latency of the PCFU to increase as well.

The number of inputs and outputs also plays a role in control generation for the PCFUs. Recall that in the PCFU control generator, the meta-register file is responsible for generating the LUT entries. For every additional input, the size of the LUTs double, meaning that the size of the meta-register file also doubles. Increasing the number of outputs is less critical for control generation, as all the LUT configurations of the live-out values are stored in the meta-register file. That is, the baseline design already supports multiple outputs, so very little additional complexity is needed to support them.

The effects of adding inputs and outputs to a PCFU can be seen in the synthesis results

in Table 3.6. In this table, each PCFU design is represented by a 4-tuple specifying the number of inputs, the number of outputs, the number of supported additions, and what shift values (if any) are supported. Initially increasing the number of inputs has a small effect on the total PCFU area and latency; moving from two to three inputs increases the area by 0.016 mm^2 and the latency by 0.21 ns. However, the exponential increase in LUT size quickly begins to dominate. For example, moving from five inputs to six causes an increase in latency of 0.22 ns, and the die area more than doubles, going from 0.214 to 0.465 mm^2. This demonstrates that the number of inputs must be carefully balanced in the design of a PCFU.

Increasing the number of outputs is not as critical of an issue in terms of PCFU design. Each additional output from the PCFU requires an additional function LUT to compute the result using the inputs and the carry-in signal(s). No additional LUTs are needed beyond that, and none of the other structures change in size. This essentially means that adding an output should increase the area of the PCFU in a roughly linear fashion, and have a small or no effect on the latency. These trends can be seen in Table 3.6. Moving from four inputs and two outputs to four inputs and three outputs increases the area by 0.059 mm^2, and the latency by 0.37 ns. The non-linearity is due to increased MUX sizes and certain signals (e.g., the carry-ins) having to drive a larger number of cells.

One confusing trend in Table 3.6, is that not all of the synthesis results agree with what was predicted. For example, moving from six inputs and one output, to two outputs, and to three outputs caused the area of the PCFU to grow super-linearly. Adding more outputs also caused the latency to change a great deal, despite the fact that the critical path has the same number of logic levels in all three designs. These observations are an unfortunate side-effect of heuristics used in the synthesis tools, and are beyond our control.

Figure 3.9 shows the average speedup across our benchmark suite when varying the number of inputs and outputs allowed in the PCFU. When adding inputs and outputs for this experiment, we assumed that reading the inputs and writing the results back to the register file each took one cycle regardless of the number of inputs/outputs. This was done to determine how well the compiler can take advantage of the inputs/outputs available to it, independent of other hardware restrictions.

The main point to take away from Figure 3.9 is that four inputs and two outputs seems to be the point of diminishing return. That is, increasing the number of inputs beyond four or the number of outputs beyond two does not substantially improve the resulting performance. Four inputs and two outputs are necessary to support the most important computation subgraphs in our set of applications. Conversely, reducing the number of inputs to three or two drops the speedup to 1.55 and 1.49, respectively. Reducing the number of outputs to one drops the speedup to 1.45. While these drops may not seem significant, the average is hiding the fact that the speedup of some benchmarks drop significantly, while other benchmarks are relatively unaffected. For example, the speedup of MD5 dropped 78% moving from four inputs to two, and the speedup of EPIC fell 58% moving from two to one output.

Number of Additions. As with the number of inputs, the number of additions supported by the PCFU must also be carefully constrained. Supporting an additional add operation would necessitate creating two new LUTs and a Kogge-Stone tree to calculate the Propagate and Generate signals for that add. These new P-G LUTs will be a function of each input and all previous carry-in signals, meaning that their size will be twice as large as the previous largest P-G LUTs. Beyond the additional LUTs, the size of each function LUT doubles, since each output is also a function of this new carry signal. This increases the area of the PCFU and lengthens the critical path much more quickly than simply adding inputs or outputs.

Adding the new P-G LUTs means that control will have to be generated for them as well. This entails adding three new registers to the meta-register file for the new P-LUT, G-LUT, and carry LUT. Space for the added control of these LUTs must be added to the configuration cache as well as the meta-register file; however, overall latency of the control generation is not affected, and area increases linearly with the number of adds.

The top portion of Table 3.7 shows the synthesis results when varying the number of adds supported. Note how increasing the number of adds supported more than doubles the die area of the PCFU in most cases. The latency of the PCFUs also increases a great deal, since the critical path now runs through an additional carry generator and larger function LUTs. For these reasons, it is important to limit the number of additions supported in the

PCFU.

One trick that can be played to reduce the area overhead of supporting many additions, is to compose a larger PCFU out of smaller ones. For example, a two-adder PCFU could be created by serially merging two one-adder PCFUs with a MUX. The MUX is used to select a subset input values from the result of the first PCFU and the subgraph inputs. Using a MUX to control the number of inputs prevents the exponential growth of the LUTs, at the cost of potentially supporting fewer subgraphs. Our experiments have shown that the subgraphs not supported never occur in any of the applications tested, though, making this a good trade off.

The lower portion of Table 3.7 shows the synthesis results of these composite PCFUs. Next to the number of adds in the design column, parentheses occur indicating the formation of the composite PCFU. For example, "(1-1-1)" indicates a three-add PCFU designed as three one-add PCFUs chained together serially. This table clearly shows how creating PCFUs as a composite of smaller PCFUs is an effective way to reduce the area incurred by supporting more add operations. The variations in latency reflect the trade off of the critical path traveling though fewer large LUTs (when the PCFU is not composite) versus the critical path traveling through more small LUTs (when the PCFU is composed of 1-adder PCFUs).

Figure 3.10 shows the effect the number of adds supported has on the speedups attainable by the PCFU system. This figure shows that significant speedup gains can be achieved by moving from zero to one to two adds. This trend highlights the prevalence of add instructions in our applications. Moving beyond two adds yields quite limited results, though. This is mainly because computation subgraphs are limited by memory operations. The amount of computation done between memory accesses typically does not encompass more than two add instructions. This is not as true in the encryption-style applications which contain a relatively large amount of computation between memory accesses.

Support for Shifts. Another design parameter explored was the addition of shift operations in subgraphs supported by the PCFU. In the general case, it is impossible to support shifts at arbitrary places within subgraphs. Doing so would make each output bit a function of each input bit creating function LUTs the size of $2^{total\ input\ bits}$. However, it is feasible to

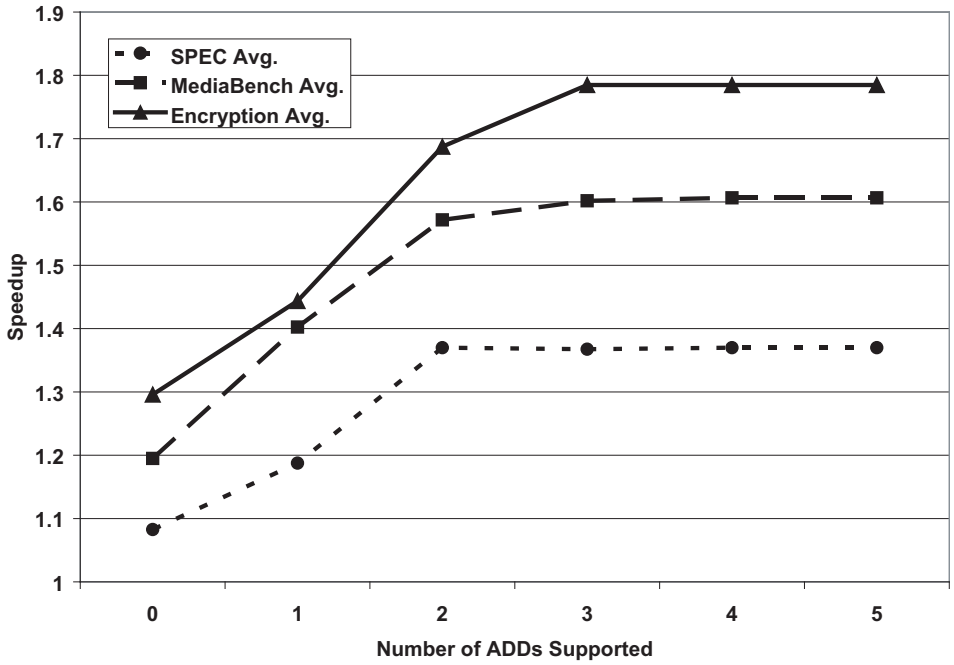


Figure 3.10: Effectiveness of PCFU designs with varying numbers of additions supported.

place a shifter at the inputs or outputs of the PCFU. This would allow support for shifts in subgraphs provided there was no computation before the shift (or after the shift, in the case of shifts at the outputs) performed on the PCFU. Adding these shifters would not affect the size of the LUTs or the internal PCFU structure, but would require some additional MUXes at the inputs (and/or outputs). The downside is that the shifters would appear on the critical path.

In terms of control generation, allowing shift capabilities in the PCFU involves adding few bits in the configuration to specify the shift value and direction for each input and/or output. Though this does increase the critical path of control generation slightly, it is a trivial extension. Allowing shift operations increases the size of the configuration size as the log of the number of shift values supported, thus the area overhead increases at that rate as well.

To analyze the effectiveness of allowing shifts within subgraphs, we first examined the types of shifts that could potentially be used. Figure 3.11 shows the types of shifts that appeared in important subgraphs. That is, if the compiler allowed shifts to appear anywhere

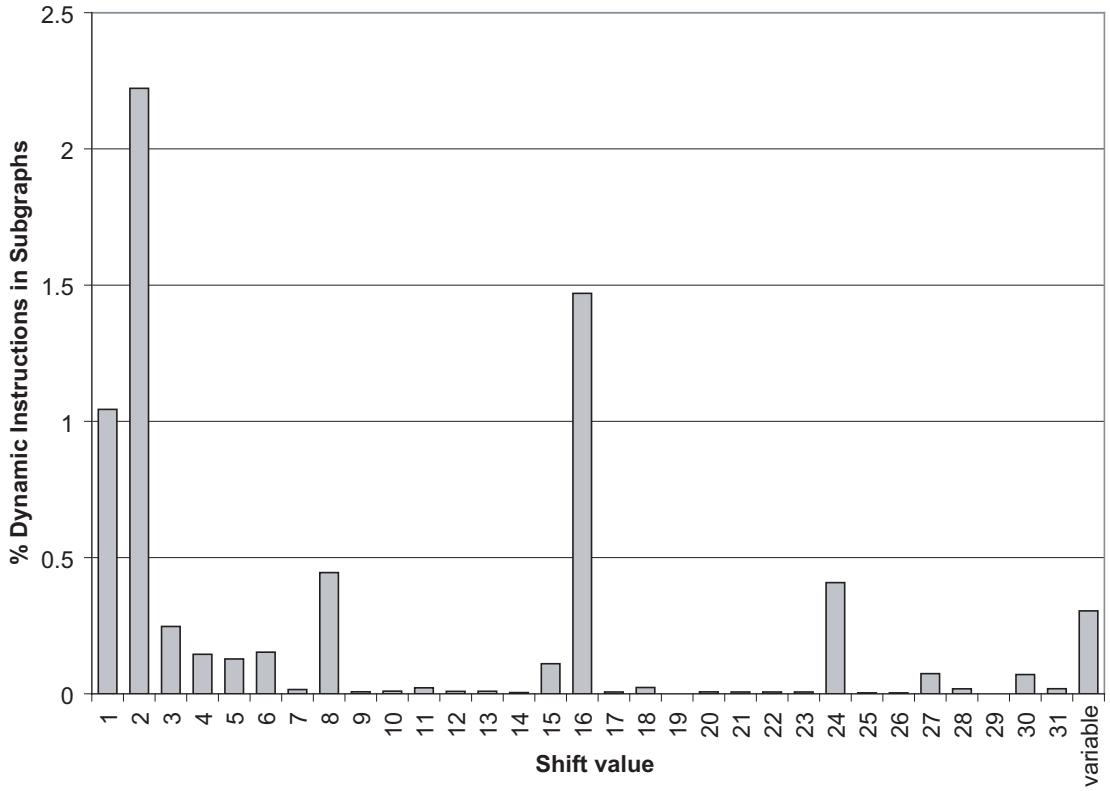


Figure 3.11: Distribution of shift values within subgraphs.

in subgraphs, this graph shows the types of shifts that were selected for subgraph inclusion; the shifts that would be useful to support in the PCFU. The horizontal axis in this figure is the constant value of the shift instruction (or variable in the case that the operation did not use a compile time constant), and the vertical axis shows the percentage of dynamic instructions averaged across the benchmarks. As an example, around 1.5% of dynamic instructions in our benchmarks were shifts by the constant 16, which would have appeared in subgraphs provided the PCFU supported them.

Figure 3.11 shows that the shifts useful in subgraphs are dominated by a relatively small number of constants. As would be expected, two is the most common shift value, since it is frequently used for address calculation in the 32-bit ARM architecture. One key trend in this figure is that variable shifts (the far right bar) were quite infrequent. This is a good sign, as supporting variable shifts in the PCFU generally requires larger area. Conversely,

Design	Latency (ns)	Area (mm^2)	Area (% ARM926EJS)
4 In, 2 Out, 2 Adds, No Shifts	4.20	0.171	7.7
4 In, 2 Out, 2 Adds, 2 at Inputs	4.64	0.213	9.6
4 In, 2 Out, 2 Adds, 1, 2, 16 at Inputs	4.86	0.224	10.1
4 In, 2 Out, 2 Adds, Any at Inputs	5.22	0.224	10.1
4 In, 2 Out, 2 Adds, Any at Outputs	5.15	0.201	9.1

Table 3.8: Synthesis results for PCFU designs with varying types of shifts supported.

supporting shifts by a small number of constants merely requires a small bit of wiring and an additional MUX.

Using this information, several designs supporting shifts were synthesized; the results are in Table 3.8. In order to limit the area overhead associated with barrel shifters, we used logarithmic shifter in the synthesized designs. The chart shows using logarithmic shifters generally caused the latency to increase a great deal when supporting additional constants, however, it did not incur a substantial area gain. For example, supporting any shift value was nearly the same area as supporting the three most frequent constants. Also note that supporting shift values at the tail of subgraphs was less costly than at the head; this is intuitive, as there are only two outputs compared with four inputs.

Speedup results for these designs are in Figure 3.12. For each set of shift values supported, there are three bars displayed: one for when shifts are supported only at the inputs (or head of the subgraph), one for shifts only supported at the outputs, and one for shifts supported anywhere within the subgraph. Although the last bar is not supported by the PCFU, it provides a comparison as to how well shifts at the inputs or outputs meet the overall need for shifts in subgraphs.

In general, providing capabilities for a small number of shift values does provide a substantial amount of speedup. For example, allowing shifts by 1, 2, or 16 at the outputs improved speedups by 7% over the baseline design. Providing shifts at the end of subgraphs is slightly more beneficial than at the head of subgraphs, again, because many shift-by-two ops are used for address calculation. The address calculation feeds memory operations, which must appear outside the subgraph. Allowing shifts anywhere in subgraphs does

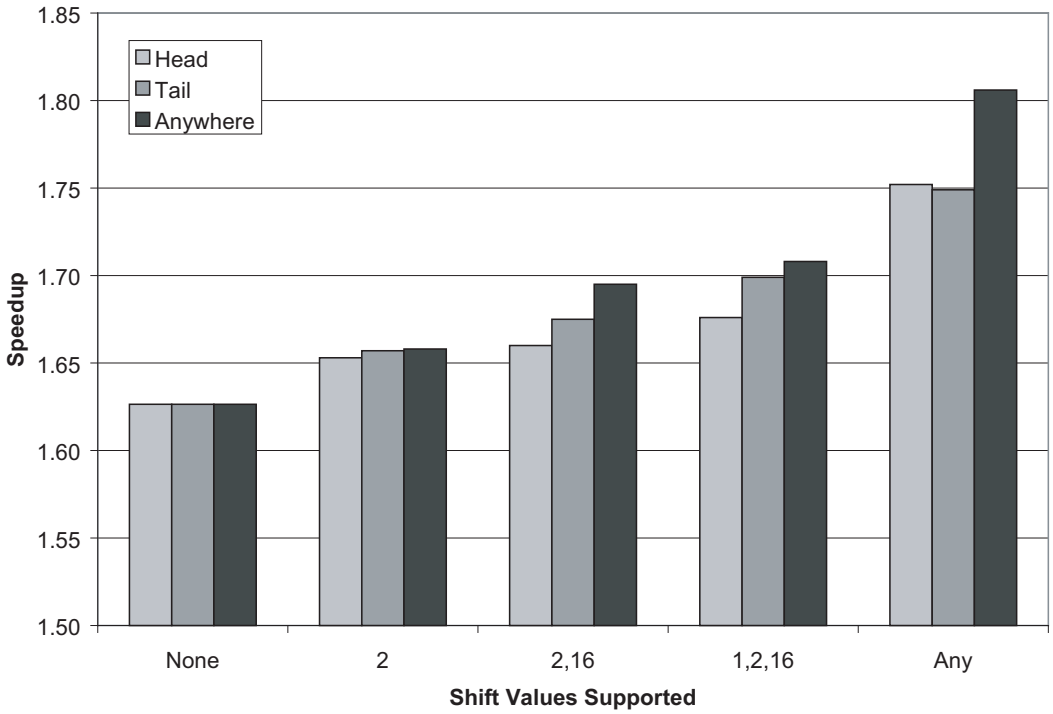


Figure 3.12: Effectiveness of PCFU designs with varying types of shifts supported.

offer significant benefit over restricting shifts to the fringes, but most of the gains from adding shifts can be attained by only adding them at the inputs and outputs.

Cost vs. Performance for all Designs. To summarize the trade offs of the design space, we combined the synthesis and simulation results in Figure 3.13. The horizontal axis has the cost of a design, and the vertical axis shows speedup attained using that design. The speedup numbers in this figure were scaled to reflect cycle time increases. The ARM926EJ-S typically runs at 250 MHz using a standard synthesis flow in 0.13μ technology [5]. In the left portion of Figure 3.13, if a PCFU design could not meet the 4ns cycle time, then the entire processor was slowed to the frequency of the PCFU. For example, if a PCFU had a critical path of 8ns, then we assumed two cycles of the baseline machine could occur in the same time as one cycle of the machine using that PCFU. The right graph in Figure 3.13 performs the same scaling, but assumes that the PCFU takes two cycles to execute (e.g., the PCFU is pipelined). This allows us to compare the cycle time versus subgraphs supported trade offs for PCFUs in the context of processors with higher clock frequencies.

The main observation to take from the 1-cycle PCFU graph is that to offset increasing the clock cycle, it is imperative to support many more subgraphs. Only one of the Pareto-optimal design points (4I, 2O, 2A, None) increased the clock cycle, and that was only by $0.2ns$. Figure 3.9 shows the increased number of subgraphs, by moving to four inputs/two outputs, needed to justify slowing the clock cycle. When clock cycle is taken into account, there generally are not enough large dataflow subgraphs to justify the PCFU designs targeting them.

Under the assumption of a two cycle PCFU, it is a different story, however. Assuming the PCFU takes two cycles to execute implies that none of the PCFU designs extend the clock cycle. This enables the benefits of supporting the larger subgraphs to show themselves. For example, the 5I, 3O, 2A, None design point is Pareto optimal under the two-cycle assumption. Despite this, using two cycles to support larger subgraphs did not outperform the one-cycle PCFU designs that target smaller subgraphs.

3.9 PCFU Summary

In the past few sections, we have explored the design of Programmable Carry Function Units, a hardware substrate for executing acyclic dataflow subgraphs. Several different design parameters were examined, ranging from the number of subgraph inputs/outputs supported, to the number of addition/subtractions supported, to the the types of shifts allowed. Evaluation of these designs was done with simulation as well as synthesis, to fully evaluate the hardware tradeoffs in the context of the ARM926EJ-S embedded processor. Overall, we have shown that implementing a carefully designed PCFU can provide substantial speedups (1.47 on average) over a baseline embedded processor for relatively little area overhead. We also demonstrated that non-pipelined PCFU designs that support more subgraphs, but increase the cycle time of the processor, are generally not wise design points. However, that conclusion is reversed in the case of pipelined PCFU designs.

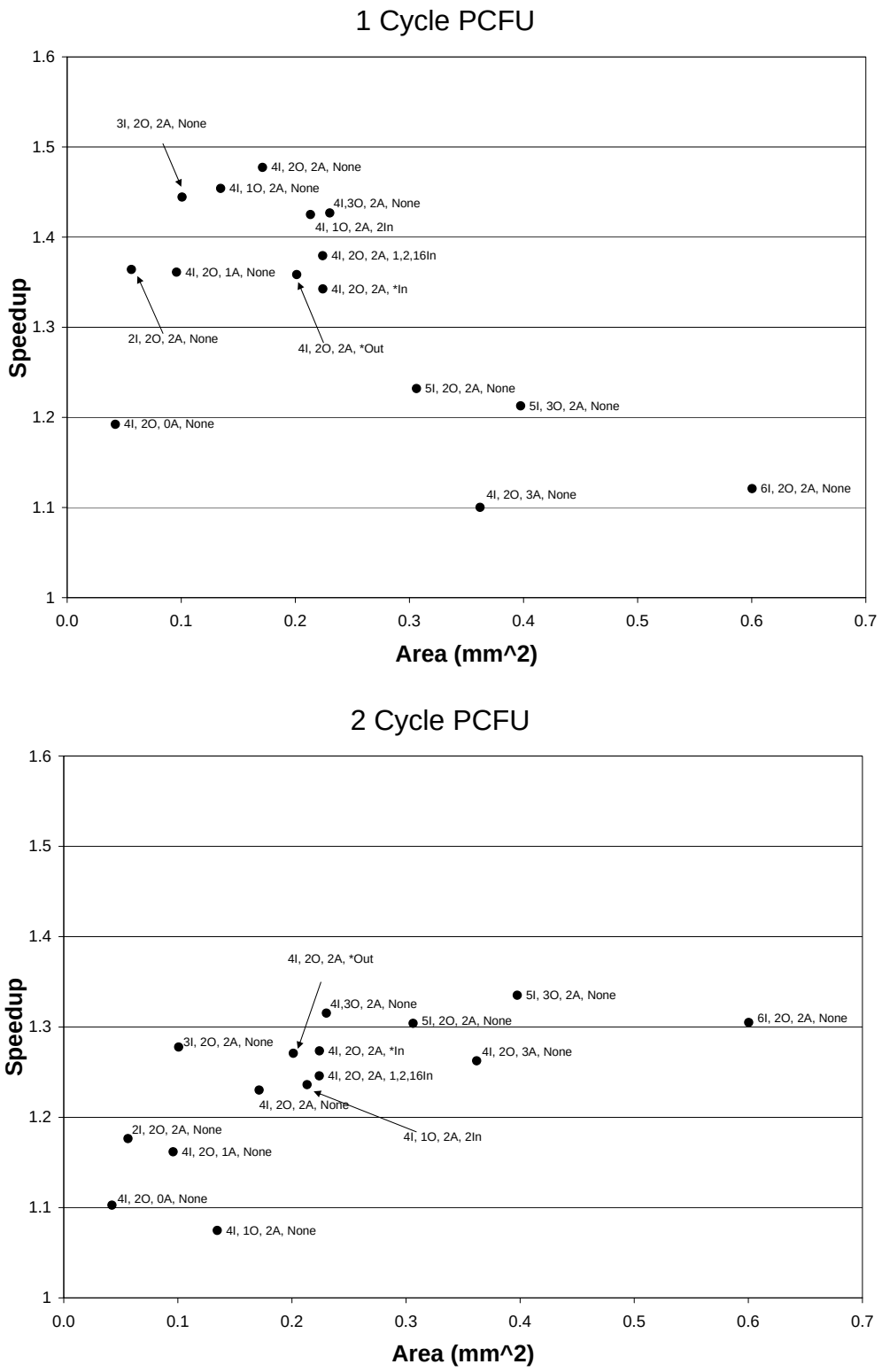


Figure 3.13: The cost/performance trade off across various PCFU design points.

CHAPTER 4

Utilization of Generalized Acyclic Accelerators

4.1 Introduction

In the previous chapter, acyclic computation accelerators were proposed that are general enough to use in a wide range of applications. In this chapter we develop ways to utilize those accelerators.

At present, specialized hardware is typically exploited through the use of customized instructions or instruction set extensions, as an application specific instruction set processor (ASIP). The central problems with an ASIP approach are the hardware design and software migration time/costs. ASIP designs incur substantial non-recurring engineering costs. For example, each new ASIP must be verified both from the functionality and timing perspectives. Additionally, a new mask set must be created to fabricate the chip. On the software side, the compiler must be retargeted to each new processor and any hand-written libraries must be migrated to the new platform. Automation of some of these tasks may be possible; however, the majority of this work is still a manual process. All of these challenges make it difficult to adopt a new ASIP despite the potential advantages.

This chapter develops two methods to utilize compute accelerators that do not alter the instruction set, thus avoiding the pitfalls of ASIPs. We term these methods *transparent instruction set customization*.

The first method proposed, a fully dynamic scheme, performs subgraph identification and instruction replacement in hardware. This technique is effective for preexisting pro-

gram binaries. The second method reduces hardware complexity using a static subgraph identification offline during the compilation process. Subgraphs that are to be mapped onto the accelerator are marked in the program binary to facilitate simple accelerator configuration and replacement at run-time by the hardware.

In the remainder of this chapter, we describe the hardware and software algorithms necessary to facilitate dynamic customization of a microarchitectural instruction stream. The tradeoffs of these algorithms are discussed and the effectiveness of each is experimentally determined.

4.2 Related Work

Once an accelerator has been designed, it becomes necessary to map portions of an application onto the accelerator. Two examples of using the compiler to statically map dataflow subgraphs onto a CCA are [32] and [101]. Both of these techniques target fixed hardware, and it is not clear if the algorithms extend to cover accelerators not exposed to the instruction set.

Several software frameworks have been proposed which would lend themselves to dynamically mapping dataflow subgraphs onto accelerators. Dynamo [10], Daisy [39], and Transmeta's Code Morphing Software [38] are all schemes that optimize and/or translate binaries to better suit the underlying hardware. These systems can potentially do a better job of mapping an application to accelerators than compile time systems, since they can take advantage of runtime information, such as trace formation. Using these systems has the additional benefits that algorithms proposed to statically map computation to an accelerator would be effective, and full binary compatibility is provided.

Many hardware based frameworks exist for this process, too. Most of these arose from the observation that in systems with a trace cache, the latency of the fill unit has a negligible performance impact until it becomes very large (on the order of 10,000 cycles [45]). That is, once instructions retire from the pipeline and a trace is constructed, there is ample time before that trace will be needed again. Three recently proposed schemes [28, 111, 130] used this latency to perform the mapping of dataflow subgraphs onto specialized execution

hardware. Instruction Path Coprocessors [25] and rePLay [42] have also propose taking advantage of this latency for other instruction stream optimizations.

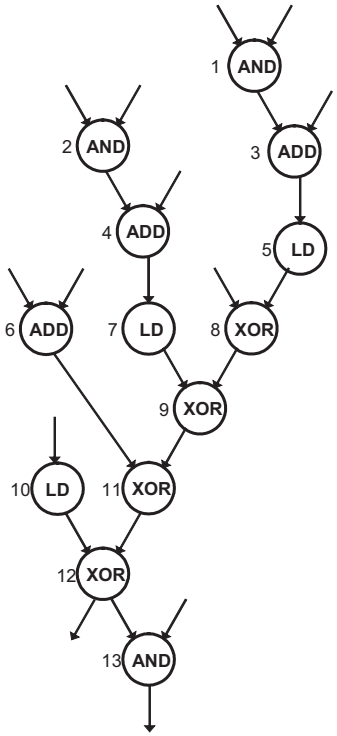
A simplified dynamic subgraph mapping system was described in [62, 113]. These papers used the design proposed in [102] as the baseline of their system, which greatly simplifies the mapping problem. Because our goal was to allow for more flexibility than their CCA design allowed for, our presented identification algorithm is much more complex.

Other recent work [17] proposes using the DISE [37] framework to dynamically replace subgraphs in the instruction stream. A special instruction is used to signal the DISE engine, which then inserts the appropriate control logic into the pipeline. This model requires a DISE aware operating system and processor, since the subgraphs are specified in the binary at load time, and must be replaced to execute the binary at runtime. Conversely, the framework proposed in this work does not affect the operating system, nor does it require any special replacement engine to run the binary.

4.3 Utilization of an Acyclic Compute Accelerator

Once an accelerator is integrated into a processor, it is necessary to provide subgraphs for the accelerator to execute. Feeding an accelerator involves two steps: *discovery* of which subgraphs will be run on an accelerator and *replacement* of the subgraphs with uops in the instruction stream. In this section, two alternative approaches for each of these tasks are presented.

The two proposed approaches for subgraph discovery can be categorized as static and dynamic. Dynamic discovery assumes the use of a trace cache and performs subgraph discovery on the retiring instruction stream that becomes a trace. When the instructions are later fetched from the trace cache, the subgraphs will be delineated. The main advantage of a dynamic discovery technique is that the use of an accelerator is completely transparent to the ISA. Static discovery finds subgraphs for an accelerator at compile time. These subgraphs are marked in the machine code using two new subgraph specification instructions, so that a replacement mechanism can insert the appropriate accelerator uops dynamically. Using these instructions to mark patterns allows for binary forward compatibility, meaning



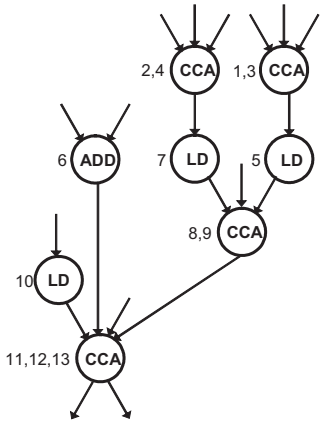
A.

Operation	Slack
1	0
2	1
3	0
4	1
5	0
6	4
7	1
8	0
9	0
10	5
11	0
12	0
13	0

B.

Step	current_match	priority_queue
1		13
2	13	12
3	12, 13	11, 10
4	11, 12, 13	9, 6, 10
5	11, 12, 13	6, 10
6	11, 12, 13	10
7	11, 12, 13	
8		9
9	9	8, 7
10	8, 9	5, 7
11	...	...

C.



D.

Figure 4.1: A. DFG from a frame in 164.gzip. B. Slack of the operations. C. Trace of Algorithm 4.1. D. DFG after subgraph replacement.

that as long as future generations of accelerators support at least the same functionality of the one compiled for, the subgraphs marked in the binary are still useful. The static discovery technique can be much more complex than the dynamic version, since it is performed offline; thus, it does a better job of finding subgraphs.

The two proposed schemes for replacing subgraphs are both dynamic, but performed at different locations in the pipeline. Replacing subgraphs in the fill unit of a trace cache is the most intuitive place for this task. As mentioned before, previous work [42] has shown that delays in the fill unit of up to 10,000 cycles have a negligible impact on overall system performance. This delay provides ample time for augmenting the instruction stream. The second proposal is to replace subgraphs during decode. The impetus behind this idea was that many microarchitectures (like the Intel Pentium IV) already perform complicated program translations during decode, so subgraph replacement would be a natural extension. The biggest advantage of a decode-based replacement is that it makes the trace cache unnecessary when used in concert with static discovery. Removing the trace cache makes accelerators more attractive for embedded processors, where trace caches are considered too inefficient and power hungry.

The primary reason for using dynamic replacement for accelerator instructions is that complete binary compatibility is provided: a processor without an accelerator could simply ignore the subgraph specification instructions and execute the instructions directly. This idea extends to future processors as well. As long as any evolution of an accelerator provides at least the functionality of the previous generation, the statically discovered subgraphs will still be effective. Essentially, this allows for binary compatible customization of the instruction set.

4.3.1 Dynamic Discovery

The purpose of dynamic discovery is to determine which dataflow subgraphs should be executed on an accelerator at runtime. To minimize the impact on performance, we propose to use the rePLay framework [99] in order to implement dynamic discovery.

The rePLay framework is similar to a trace cache, in that sequences of retired instruc-

```
1  for i = N to 1 do
2      if op_i is in a match then
3          Continue
       end
4      Initialize current_match
5      priority_queue.push(op_i)
6      while priority_queue not empty do
7          candidate_op ← priority_queue.pop()
8          Add candidate_op to current_match
9          if current_match does not meet constraints then
10             Remove candidate_op from current_match
11             Continue
           end
12         foreach parent of candidate_op do
13             if parent is not in a match then
14                 priority_queue.push(parent)
               end
           end
       end
15     if accelerator implementation of current_match is better than native implementation
       then
16         Mark current_match in instruction stream
       end
17     current_match.clear()
   end
```

Algorithm 4.1: Dynamic discovery algorithm

tions are stored consecutively and later fetched. RePLay differs because instead of traces, it uses frames, where highly biased branches are converted into control flow assertions. A frame can be thought of as a large basic block, with one entry and one exit point. If any of the control flow assertions are triggered, the entire frame is discarded. This property of rePLay provides an excellent opportunity for subgraph discovery, since subgraphs are allowed to cross control flow boundaries without compensation code. A frame cache also allows for ample time between retirement and when the instruction stream will be needed again.

The algorithm proposed for dynamic subgraph discovery and selection is shown in Algorithm 4.1. The basic idea underlying this algorithm is to start at an operation not already

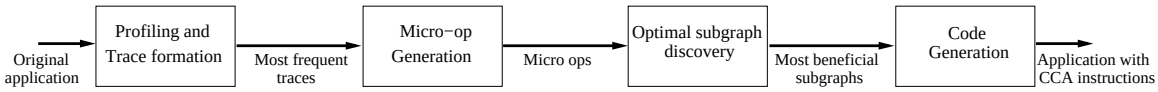


Figure 4.2: Workflow of static discovery

in a match, and then grow that seed operation toward its immediate parent operations. When parent operations are added to the seed operation, a new subgraph is created for replacement, provided that the subgraph meets the architectural constraints of an accelerator. These constraints include number of inputs/outputs, illegal opcodes, and subgraph outputs cannot feed subgraph inputs (necessary to avoid deadlock). An operation's slack (i.e., how critical each operation is to the total dependence height of the DFG) is used to determine the priority of adding operations to the match when multiple parents exist. This heuristic is reminiscent of both Dijkstra's shortest path algorithm or the 'maximal munch' code generation algorithm.

To better illustrate Algorithm 4.1, Figure 4.1C shows a sample run on the DFG in Figure 4.1A targeting the height-four CCA from from Table 3.4. The discovery algorithm starts at the bottom operation of the frame with operation 13. Node 13 is popped and added to the match at step 2. Next 13's parent, node 12, is added to the queue and subsequently to the current match. When 12's parents are added to the queue in step 3, note how 11 is ahead of 10 in the queue because it has a slack of 0 as compared to 5. Slacks for all operations are given in Figure 4.1B. At step 5, node 9 would be added to the match; however, the resulting subgraph would require 5 inputs, which violates the architectural constraints of the accelerator. Node 9 is simply discarded and its parents are ignored. This process continues until the priority queue is empty at step 7 and a subgraph is delineated. After the subgraphs are replaced, Figure 4.1D shows the resulting DFG.

This heuristic guides growth of subgraphs toward the critical path in order to reduce the dependence height of the DFG. The reason subgraphs are only grown toward the parents of operations is because this reduces the complexity of the discovery algorithm, and it guides the shape of the subgraphs to match the triangular shape of the proposed CCA design.

Note that this algorithm is just a greedy heuristic, and will not perform as well as offline discovery algorithms that have been developed.

4.3.2 Static Discovery

In order to reduce the complexity of the hardware customization engine, a method for offline customization of applications is also proposed. This approach builds on traditional compiler-based techniques for instruction set customization, and is shown in Figure 4.2. Initially, the application is profiled to identify frequently executed frames. If the execution engine uses microcode, the compiler converts the frames from sequences of architectural instructions to sequences of uops to match what would be seen by the replacement engine. The most frequently executed frames are then analyzed and subgraphs that can be beneficially executed on an accelerator are selected. Then, the compiler generates machine code for the application, with the subgraphs explicitly identified to facilitate simple dynamic replacement.

Trace formation: A trace is a sequence of basic blocks that are highly likely to be executed sequentially [85]. Traces are identified by profiling the application on a sample input. The trace structure is very similar to the frames that are identified and optimized by rePLay, thus the compiler uses traces as a surrogate for the frames formed by the hardware.

Micro-operation generation: In order to identify subgraphs that can be replaced at run-time, the compiler must convert its internal representation to match the run-time instruction stream. For instruction sets such as x86, this implies converting instructions into micro-operations, thereby creating a uop trace. The compiler also keeps track of mapping between instructions and uops to facilitate later code generation. When targeting microarchitectures without uops, this step is unnecessary.

Subgraph discovery: The subgraph discovery algorithm used for this chapter is based on previous two works [32] and [7]. As described in [32], the subgraph discovery can be logically separated into two phases: (a) candidate enumeration, that is enumerating the candidate subgraphs that can be potentially become an accelerator instruction, and (b) candidate selection, that is selecting the beneficial candidates. The branch and bound technique

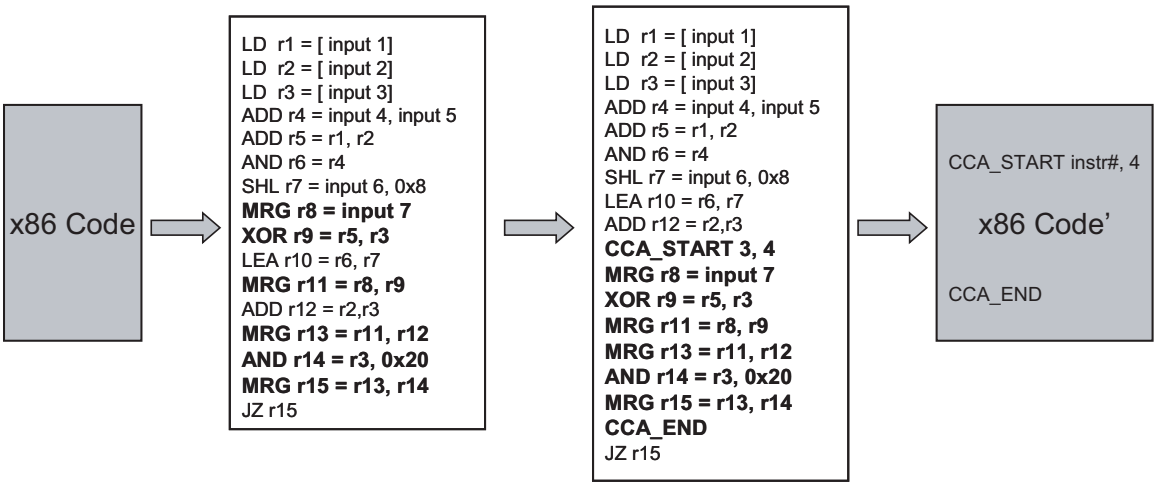


Figure 4.3: Static accelerator instruction insertion

similar to [7] was used to solve the first phase. One additional constraint was added so that all micro-operations for a particular instruction should be included in the subgraph. The selection phase was modeled as an instance of the unate covering problem. All nodes in the DFG corresponding to the trace under consideration have to be covered by the candidate subgraphs so that the overall performance is maximized. The ratio of number of nodes in the original DFG to the number of nodes in the DFG with candidate subgraphs replaced with accelerator instructions was used as the performance metric. An additional weight was given to nodes based on their slack so that subgraphs on the critical paths are more likely to be selected.

Code generation: After the best subgraphs to execute on an accelerator have been identified, the compiler must generate machine code for the application. The objective of this process is to organize the machine code in a manner that facilitates simple dynamic replacement of the uops by accelerator control signals. To accomplish this two new instructions are introduced into the ISA: $ACCEL\_START(liveout, height)$ and $ACCEL\_END$. ACCEL_START and ACCEL_END serve as markers for the instructions that comprise a subgraph to be mapped onto an accelerator. ACCEL_START has two operands: liveout is the number of the uop that produces an externally consumed register value, and height is the

maximum depth of the micro-operation subgraph. Note that the last uop of the subgraph is assumed liveout, creating a maximum of two outputs. Height is used as a quick feasibility test to efficiently support multiple accelerator variations.

For each uop subgraph, the code generator groups the corresponding macro-instructions together. The assembly instructions are topologically sorted based on the structure of the subgraph and placed sequentially in memory. an ACCEL_START instruction is pre-pended to the list and an ACCEL_END is post-pended, thereby isolating the subgraph and making it simple for the hardware to discover. For any case where a accelerator enabled binary needs to run on a processor without an accelerator, the ACCEL_START and ACCEL_END instructions are converted to NOPs.

The code generation process is illustrated in Figure 4.3, which is the x86 instruction / micro-operation view of a DFG from the SPECInt benchmark crafty. The initial trace of x86 code is shown on the left, which is then converted into micro-operations as shown in the second box. A subgraph to be mapped onto an accelerator is identified as shown by the darker uops. The code generation process groups the micro operations contiguously, topologically sorts them and inserts the ACCEL_START and ACCEL_END operations as shown in the third box. The sequence of micro-operations is then mapped back to augmented x86 instructions that contain the sorted instructions together with the accelerator instructions, thereby identifying the micro-operation subgraph at the instruction level.

4.3.3 Subgraph Replacement in Retirement

Replacement is the final step in making use of an accelerator, consisting of generating the encoding bits for a given subgraph and substituting them into the instruction stream. As mentioned in Section 3.3, the encoding of accelerator instructions specifies the opcodes for each node of an accelerator and the communication between each of the nodes. Determining the communication of nodes requires one top-down pass over the operations to determine producer/consumer relationships. Placing individual operations at nodes in an accelerator can also be done with one pass over the operations by placing each node in the highest row that can support the operation while honoring data dependencies. In the

case where back-to-back additions are needed, but not supported by an accelerator, move operations are inserted to pass data from the first addition to the second.

As mentioned previously, the rePLay pipeline is an excellent place to perform subgraph replacement for an accelerator. Taking advantage of frames allows the replacer to create subgraphs that cross control flow boundaries. Additionally the latency tolerance of a frame cache allows ample time for replacement to take place.

4.3.4 Subgraph Replacement in Decode

The other alternative is to replace subgraphs during decode. This technique has smaller hardware overhead - as the frame cache is unnecessary - but decode-based schemes are more sensitive to latency and do not allow subgraphs to cross basic block boundaries.

One possible solution to the latency issue is to take the burden of generating control bits for accelerator instructions out of the decode stage. To accomplish this, we propose allowing a certain number of subgraphs to be predefined in the binary and saved into a translation table when an application loads. The ACCEL_START instructions could then just store a pointer into this table for the encoding bits, making replacement trivial. The obvious benefit is that this scheme has very low hardware overhead. However, there is an additional constraint that the number of subgraphs that can be used for the entire program is limited by the size of the translation table.

4.4 Experimental Evaluation

The proposed discovery and replacement schemes were implemented in SimpleScalar [9] using the ARM instruction set. The machine configuration for these experiments was identical to the 4-issue processor with CCAs used in Section 3.4. Within SimpleScalar, some ARM instructions are broken into micro-operations, e.g., load multiple, which performs several loads to a contiguous sequence of addresses. Many ARM instructions allow for an optional shift of one operand, and it is important to note that these shifts are also broken into uops. Since our CCA does not support shifts, it would otherwise not be possible

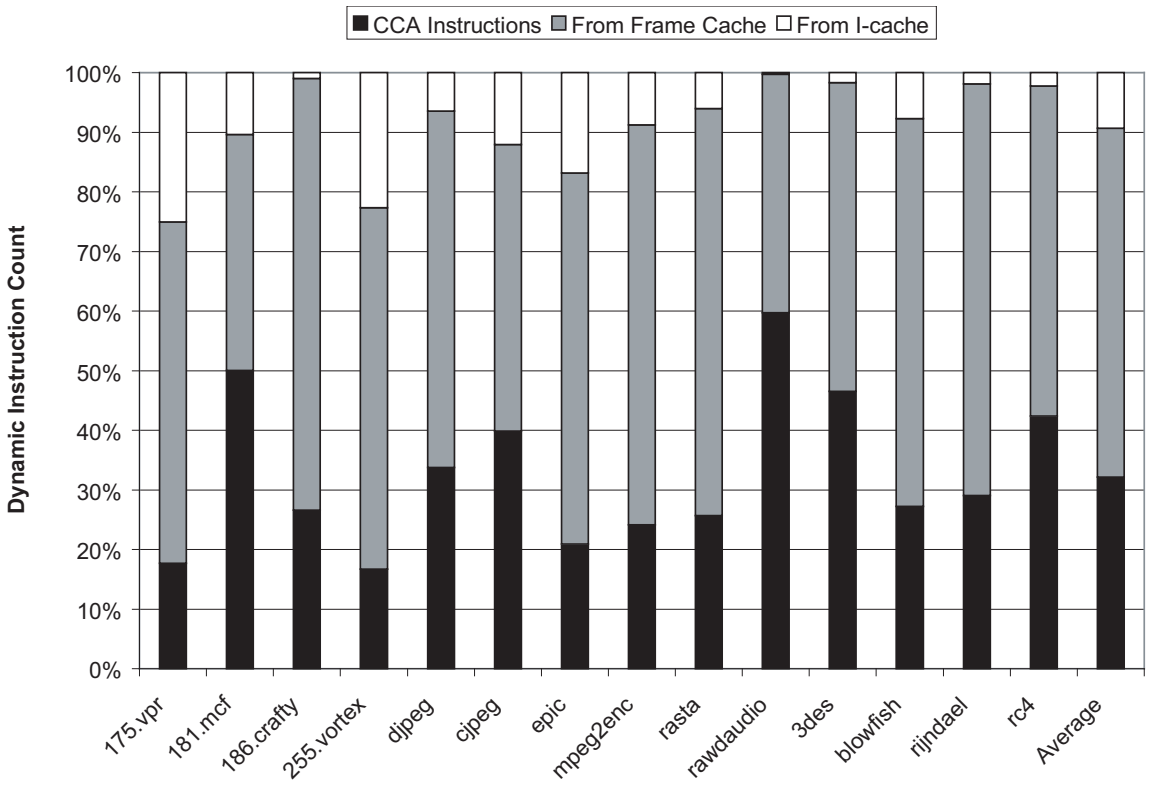


Figure 4.4: Percentage of dynamic instructions from the I-cache and frame cache

to execute these operations on the CCA.

Figure 4.4 shows the breakdown of instructions executed by the processor. The combined gray and black portions of the bars represent the percent of dynamic instructions that were provided by the frame cache. The black portion of the bars represents the fraction of dynamic instructions that were executed on the CCA. When using retirement based replacement schemes, it is very important to achieve high coverage, since CCA instructions only appear in the instruction stream from the frame cache. On average, 91% of instructions came from the frame cache in our simulations. The static discovery/retirement based replacement scheme was able to replace 35% percent of the frame cache instructions (or 32% of the total dynamic stream) with CCA operations.

As expected, a larger fraction of replaced instructions generally leads to better attained speedups. For example, 3des and rawdaudio both have a high percentage of their instruc-

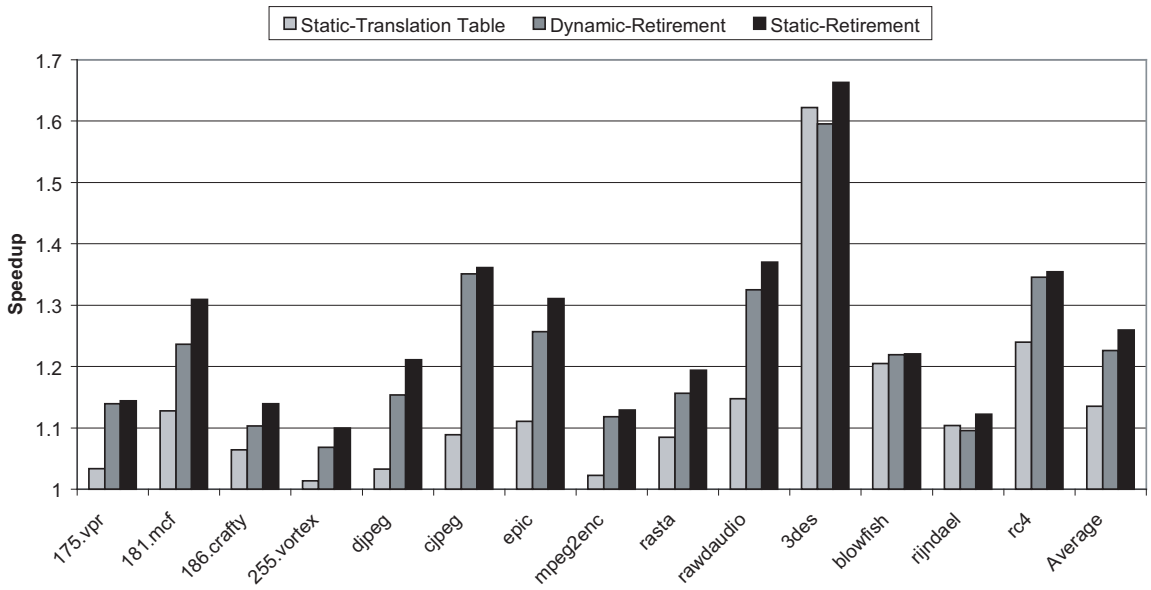


Figure 4.5: The effect of various discovery/replacement strategies

tions executed on the CCA, and they are among the applications with the highest speedups in Figure 3.2. However, there is not a one-to-one correspondence between CCA coverage and speedup. Since many replaced subgraphs may not appear on the critical path, their acceleration will only have a small impact on program execution time.

A second experiment is presented in Figure 4.5, comparing the three different discovery and replacement strategies on processor performance. The first strategy employs static offline pattern discovery and relies on a translation table in decode to replace instances in the instruction stream. The second strategy performs dynamic discovery and replacement in the fill unit of the frame cache. The third strategy is static discovery with replacement done in the fill unit of the frame cache. All three of these strategies were run using the depth 4 CCA. A translation table size of 32 was chosen for the static-translation table strategy, because previous work [117] showed that only marginal increases ($<$0.5%) in dynamic coverage are possible beyond 20 patterns.

The most apparent trend in the graph is that the static-translation table strategy typically does rather poorly when compared against the other two techniques. Investigation showed that this was not because of a limited number of available subgraphs. Rather, this method

lacks a rePLay-style mechanism to roll back the processor state, which effectively allows subgraphs to span control flow boundaries. When any branch in a frame is mispredicted, an assertion occurs and the frame is discarded. Therefore, the frame can be treated as a large basic block for subgraph replacement. Without the rePLay mechanism, it is more difficult to allow subgraphs that execute on the CCA to span control flow boundaries. For this study, we conservatively do not allow any CCA subgraphs to span a branch. While this approach is correct, a large number of potential CCA subgraphs are lost with this method. Future work includes relaxing this constraint which will likely increase the effectiveness of the static-translation table.

The graph also shows that, as expected, the static discovery outperforms dynamic discovery with the frame cache. This is because the static scheme is using a much more powerful discovery technique than the simple dynamic heuristic. However, the dynamic heuristic does do quite well in a number of cases: 175.vpr, cjpeg, and rc4. One reason for this is the underlying ISA. Since the ARM ISA has only 16 architecturally visible registers (and several are reserved), the compiler often inserts a large number of loads and stores into the code for spilling. Since the CCA cannot execute memory operations, the spill code artificially limits the amount of computation in the dataflow graph. Larger amounts of computation generally results in more options during subgraph discovery, implying that the dynamic discovery algorithm is more likely to have its sub-optimality exposed. The difference between static and dynamic discovery strategies is likely to be more pronounced with an ISA that supports a larger number of registers and thus exposes more of the true data dependencies.

4.5 Transparent ISA Customization Framework for Embedded Processors

The beginning of this chapter looked at three different techniques for transparent instruction set customization. The first, a dynamic method was proposed to identify and remap subgraphs to accelerators in a trace cache fill unit [99]. The second, a static strategy

identifies subgraphs offline during compilation and replaces the subgraphs with accelerator instructions at run-time using a decode time translation table provided in the binary. Both solutions have major drawbacks. The dynamic approach relies on a trace cache and its associated hardware optimization system. Such hardware is generally not appropriate for embedded processors due to cost and energy consumption. Further, run-time identification of patterns is inherently constrained to simple approaches as it is performed during application execution. The static approach offers no flexibility in terms of supporting multiple accelerators, as a fixed mapping to the CCA is assumed. Further, register encoding limitations in the general purpose processor (GPP) instruction set severely restrict the size of subgraphs that can map to the CCA.

The next few sections go into detail of the third approach: static selection of subgraphs with replacement at retirement time. We present how this method provides an *architectural framework* to efficiently support transparent instruction set customization in an embedded general purpose processor, such as an ARM. Subgraphs targeted for acceleration are identified during compilation or as a post-link optimization and are marked in the program executable. At run time, subgraphs are discovered, mapped, and executed on specialized hardware blocks. The hybrid approach enables the combination of sophisticated offline subgraph detection algorithms with the flexibility of online realization of the customized instructions.

Several important challenges are addressed in the proposed framework. First, a plug-and-play accelerator model is defined that consists of an augmented GPP pipeline with a predefined interface to an optional hardware accelerator block. The augmented GPP is designed and verified once. Second, the framework supports a wide range of accelerator designs including standard predefined accelerators (such as a CCA) and user-defined hardware accelerators. Regardless of the specific accelerator (or lack thereof), a single application binary is created and executed on all platforms. Third, the acceleration of complex acyclic computation subgraphs is supported. Prior work often limits subgraphs to linear chains, thereby precluding many of the performance benefits achieved with custom instructions in ASIPs. Fourth, the limited expressibility of the target instruction set architecture in terms of register names does not limit contents of the selected subgraphs. For

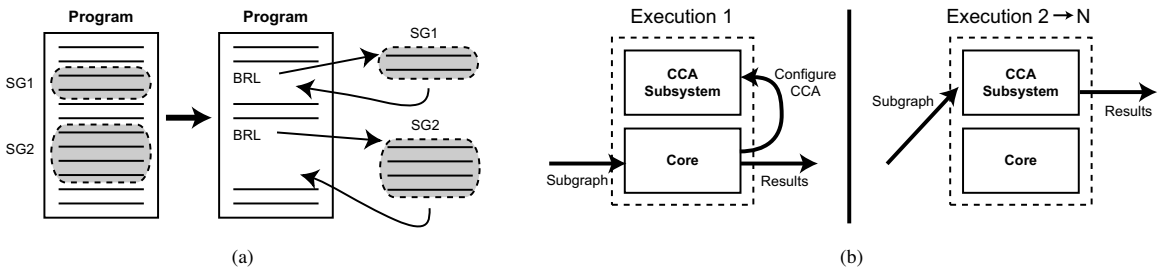


Figure 4.6: A high-level overview of the executing with a CCA: (a) subgraph identification and relocation and (b) setting up the CCA subsystem on the first invocation of a subgraph for future uses

GPP instruction sets such as ARM with only 16 registers, register spills often limit subgraphs to small sizes, thus its important to overcome this limitation. Finally, a low-cost and energy-efficient solution is selected to make the approach appropriate for embedded computing.

The key benefit of this framework is that it provides a clean interface between a processor pipeline and an accelerator, enabling easy customization of accelerators for an expected system workload. We demonstrate how the framework can process a dataflow subgraph to generate accelerator instruction on the fly, without the costs associated with a trace cache. Beyond the architecture framework, we also describe the compilation process, by which subgraphs are identified in applications and communicated to the architecture framework.

4.6 Architectural Framework

The primary contribution of this work is a configurable architectural framework to facilitate transparent instruction set customization. This framework allows architects to design hardware accelerators tuned for an expected workload and easily incorporate them into a general purpose processor via a well-defined interface. The use of a workload-specific accelerator allows manufacturers to build machines targeted toward many domains at the cost of designing and verifying only a single general purpose core and a set of applicable accelerators.

This section begins with an operational overview of the framework. The remaining subsections present a description of the proposed pipeline microarchitecture, the stages of execution of dataflow subgraphs within this pipeline, and the system interface to support subgraph execution on the system.

4.6.1 Overview

The objective of a framework for transparent instruction set customization is the support of a hybrid form of execution where subgraphs are statically identified and dynamically realized. Static identification refers to offline compiler identification of potential subgraphs for execution on custom hardware. Dynamic realization refers to hardware synthesizing the custom instructions at run-time and offloading their execution to the CCA.

The high-level process is illustrated in Figure 4.6. Initially, a program is analyzed by the compiler to identify critical computation graphs that can be mapped onto the CCA. The operations that comprise the subgraphs are pulled out of their original locations and placed into a separate function body as illustrated in Figure 4.6(a). The BRL, or branch and link, instruction is used to denote a function call in this figure. Dynamic realization is accomplished in two phases. Initially, the subgraph is executed on the hardware of the uncustomized core, denoted as Execution 1 in Figure 4.6(b). During this execution, a hardware engine determines the CCA configuration necessary to execute the entire subgraph as an atomic unit. In essence, a complex opcode is synthesized on the fly. On subsequent executions of the subgraph, the new complex opcode is substituted for the invocation of the subgraph function. Thus, as shown in Figure 4.6(b), the standard hardware must execute the first occurrence of the subgraph, while all subsequent executions will be relegated to the CCA.

The combination of static identification and dynamic realization enables powerful offline algorithms to optimize code for subgraph extraction. Further, a well-defined architectural interface introduces a layer of flexibility so that previously designed and verified cores can be easily integrated with multiple CCA designs. The remainder of this section expands the details of the architectural framework to accomplish this model of execution.

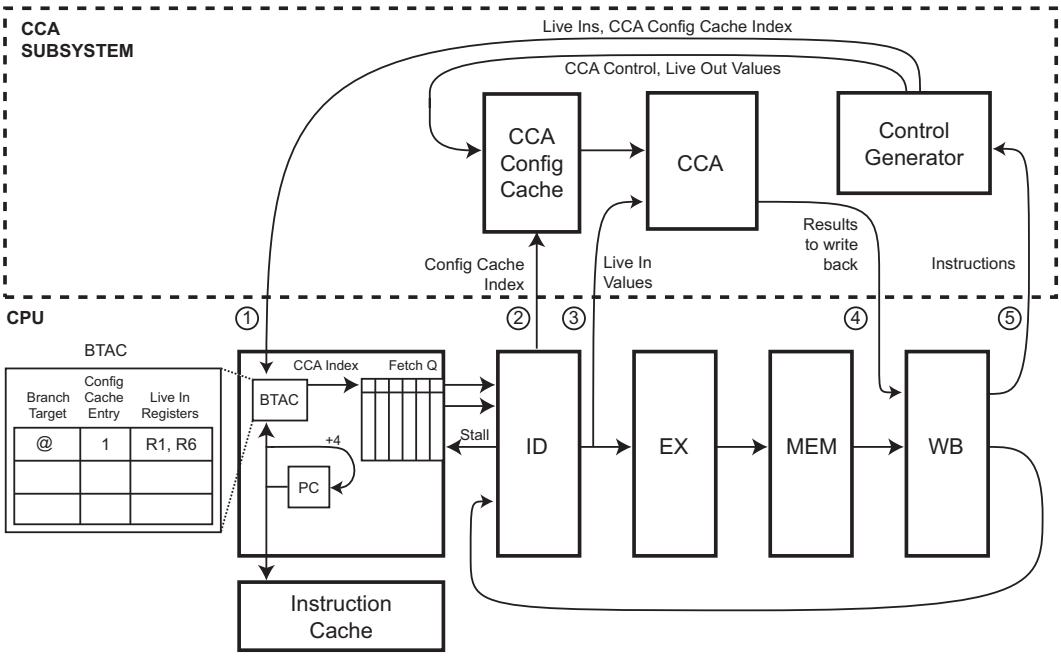


Figure 4.7: Transparent instruction set customization architectural framework

4.6.2 Pipeline Organization

Figure 4.7 presents a block diagram of the proposed architecture framework. The baseline processor, at the bottom of the figure, is augmented with the CCA subsystem at the top of the figure. The CCA subsystem consists of three major parts: the CCA itself, a configuration cache, and a control generator. The control generator is responsible for examining a sequence of retiring instructions and determining the required control signals for the CCA. Each entry of the configuration cache specifies the necessary control signals for configuring the CCA, including the opcode implemented on each CCA function unit, the interconnect between function units, and any literal values used by the subgraph.

The core processor is augmented in several places to interact with the CCA. Changes primarily occur in the instruction fetch stage of the pipeline, where instruction stream substitution occurs. The branch target address cache, or BTAC (sometimes called BTB in other literature), is extended to store additional information to decide when it is possible to substitute a CCA instruction for an invocation of a subgraph function. To accomplish this, a CCA configuration cache entry and register indexes for values consumed by the subgraph

are included in the BTAC. The decode and writeback stages are also modified to provide register inputs and accept register results from the CCA.

Central to the framework is a well-defined interface between the core and the CCA subsystem. The interface is designed so that the core can use multiple CCA designs. Since any hardware placed on the CCA subsystem increases the cost of customization, the necessary structures were integrated into the main pipeline as much as possible while maintaining the flexibility of the interface. The numbered arrows in Figure 4.7 denote the five interface points between the CCA subsystem and the CPU. These points are the only communication required between the CPU and the CCA subsystem:

1. The CCA subsystem generates entry information for the BTAC. This includes subgraph live-in register indexes and a configuration cache index where the control bits are stored.
2. During instruction decode, the configuration cache index is sent to the CCA subsystem.
3. As previously mentioned, the decode stage also provides the CCA with values for registers that are inputs to the subgraph.
4. The output values from the subgraph are relayed from the CCA subsystem back to the CPU for register writeback.
5. After retirement, completed instructions are provided to the control generator so that it can synthesize the CCA instructions from dataflow subgraphs.

4.6.3 Dataflow Subgraph Execution

A single instruction is added to the baseline instruction set to allow the compiler to delineate patterns for execution on the CCA hardware. A discussion of how the compiler uses these instructions follows in Section 4.7. The introduced instruction is dubbed BRL' because its semantics are very similar to a branch-and-link operation commonly used for subroutine calls. BRL' is treated just like a normal branch-and-link instruction in processors without a CCA subsystem: the current program counter (PC) is stored to a link register and

control branches to the branch target address. The processor without a CCA will execute the instructions in the target subroutine and return to the call site, just as it would for any other subroutine. To a processor with a CCA subsystem, the BRL' signifies the start of a subgraph to execute on the CCA.

When the BRL' is fetched from the instruction cache, its address is used to index into the BTAC. The BTAC is a standard component of modern branch prediction schemes used to hold the destination of a taken branch. In this framework, the BTAC is augmented to contain two additional pieces of information for each BRL' instruction. Register numbers for the inputs to CCA instructions are one of the additional pieces of information. These values are fed to the instruction decode stage for register reads. An index into the CCA configuration cache is the second additional piece of information stored in the BTAC. The configuration cache on the CCA subsystem contains the control bits for the CCA execution unit. If a BRL' hits in the BTAC, the configuration cache index is passed through the pipeline with other control bits and the PC simply increments to the next instruction (i.e., the branch is not taken because the BRL' was recognized as a subgraph). This prevents pipeline bubbles that would form if the branch target was taken. If the BRL' misses in the BTAC, then it is executed as a normal BRL and control branches to the procedure.

Recall that control bits from the BTAC provide the registers that are read during the decode stage of execution. Since we assume only two register reads are supported in one cycle, it may be necessary to use multiple cycles to read all of the operands necessary for the CCA instruction. Extra communication is provided allowing the decode stage to stall the fetch unit in order to facilitate this multi-cycle register read. As the registers are read, they are passed to the CCA system, keeping the width of the interface connection to a minimum.

The BTAC also passes a configuration cache index through the decode stage and into the CCA system. The configuration cache contains information pertaining to the routing of the signals on the CCA, as well as the operations to perform at each node in the CCA grid. This information is separated from the BTAC for two main reasons. First, the number of control bits is highly dependent on the structure of the CCA. Putting the configuration cache in the core, as part of the BTAC, effectively restricts the size and organization of the

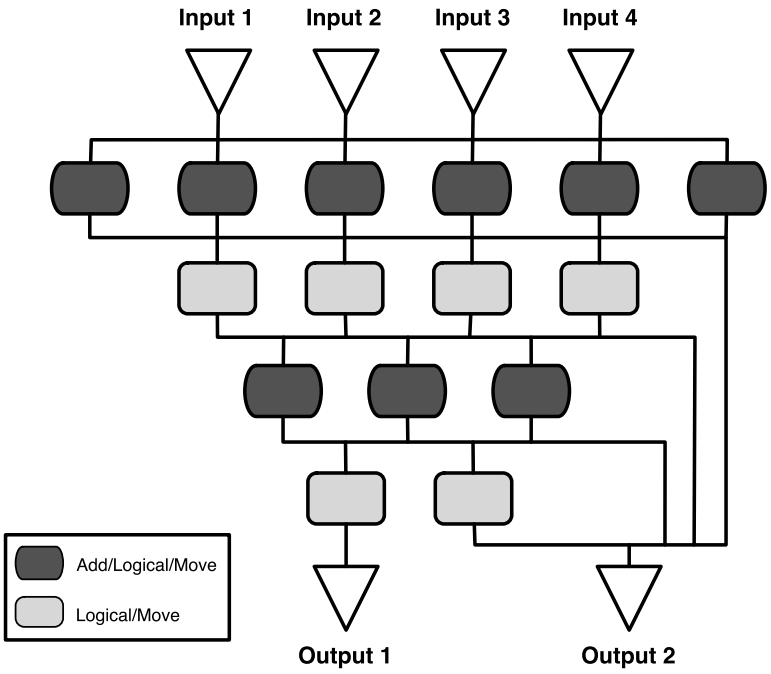


Figure 4.8: Example of a CCA implementation

CCA, since the number of control bits is set a priori. Second, putting the control bits in a separate configuration cache allows reuse of the same control bits for different subgraphs. For example, if two separate subgraphs were identical except for the registers that provide their inputs, they could share an entry in the configuration cache.

Once the registers and configuration data are passed along, the CCA executes the subgraph as a single operation and feeds the results to the writeback stage of the core. The CCA operates like any other function unit in this regard. An example of a potential CCA implementation can be seen in Figure 4.8. The CCA here is implemented as a grid-like grouping of function units with full interconnect between adjacent rows. Because of delay constraints, the two rows have slightly different opcodes available for execution, the white nodes support add, subtract, compare, sign extend, and all logical operations, while the gray nodes only support sign extend and logical operations. The design in this figure was taken directly from our previous work [28], and a more thorough discussion of the design rationale is described there. After execution on the CCA, results are written to the register file and instructions are fed the the CCA control generator, which is responsible for

mapping subgraphs onto the CCA.

4.6.4 Dataflow Subgraph Control Generation

Dynamically determining the control signals for the CCA is the most complex portion of the CCA subsystem, and is best illustrated through an example, as shown in Figure 4.9. In this example, the subgraph in the top left corner will be mapped to the CCA in the bottom left corner. The nodes of this CCA are labeled A-O for easy reference. The assembly code and subgraph in this example were taken from the Rijndael encryption algorithm.

Instructions are fed through the control generator one at a time after the writeback stage. The two loads at the top of the example are fed through and ignored, since they are not part of a subgraph. When the third instruction, a BRL', is retired, it signals the beginning of a subgraph and that the CCA subsystem should generate control information for it. The PC of the BRL' is recorded so that it can be used to update the BTAC with the appropriate data when the subgraph has been fully processed.

After the BRL', instructions are mapped to the CCA grid as they enter the control generator. Determining where to map the instructions requires several pieces of state, shown in the right portion of Figure 4.9. The table at the top of each step is a content addressable memory, or CAM, that maps a stack offset to a node that produces the value. The CAM is used to determine which node in the CCA produced the spilled value when a different operation in the gets its input from the stack. This allows the control flow generator to eliminate spill code of transient values within the subgraph. The size of this CAM equals the number of nodes in the CCA, since each node could potentially spill its produced value.

Since the proposed CCA subsystem does not support memory access operations, if the compiler is unable to allocate registers to all of the transient variables in a subgraph, then spill code would effectively partition the subgraph. This restricts performance improvement simply because of register pressure and is our rationale for performing spill code elimination.

The second piece of state in Figure 4.9 is the current producer table. For each register in the machine, this table contains the node of the CCA that produced the most recent

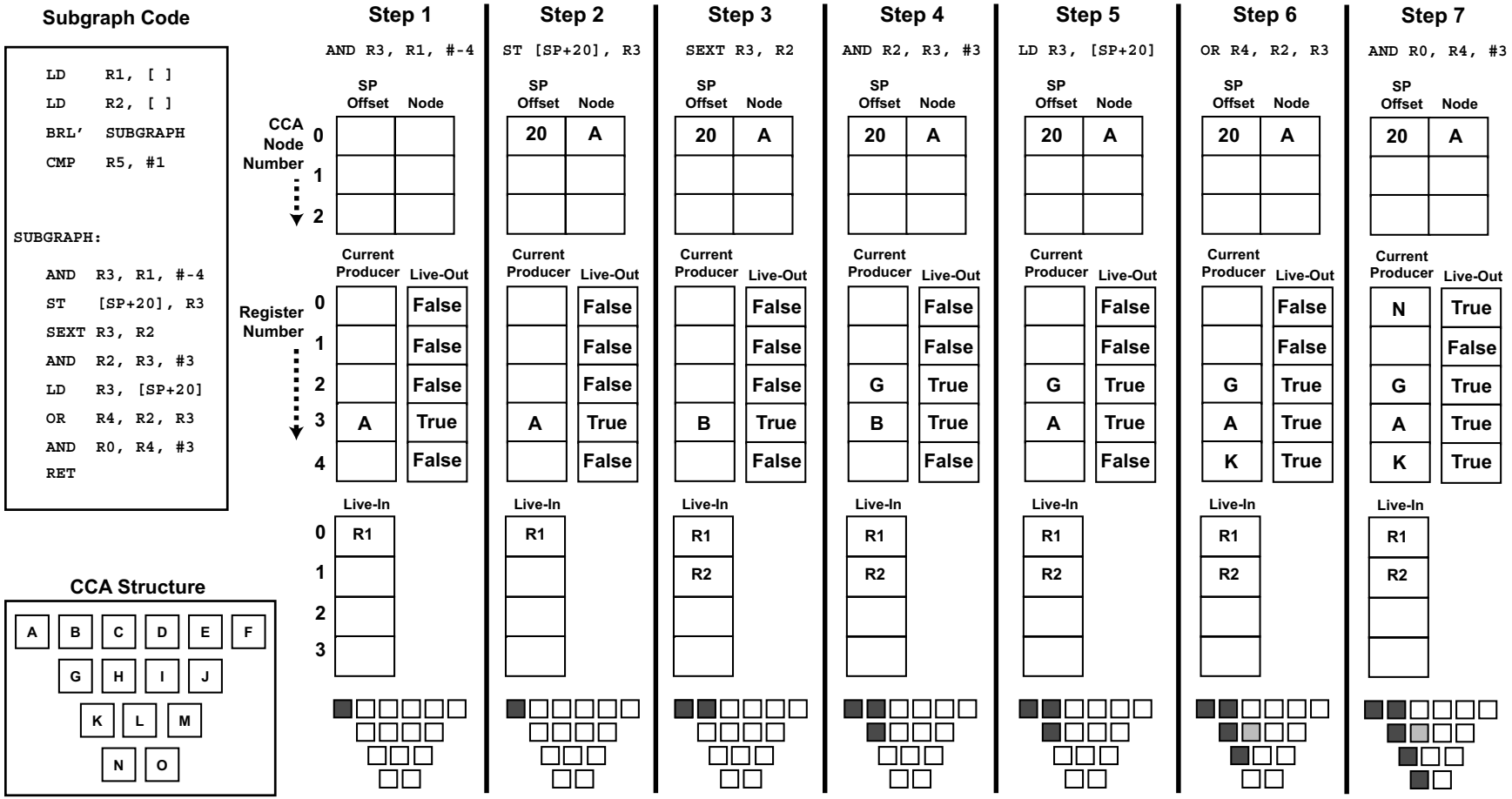


Figure 4.9: Example mapping subgraph onto a CCA

value computed for that register. The control generator also keeps two tables marking live-in and live-out values of the current subgraph. The table of live-out values records every time a value is produced by a CCA node. It is necessary to assume that all register values created are live-out, and must be written to the register file, until proven otherwise. The live-in registers record which registers are needed as inputs to the subgraph and are communicated to the BTAC after control generation is complete. The live-in table is the size of the maximum number of inputs allowed on the CCA execution unit, in this case four. At the bottom of each step is a running count of the nodes in the CCA (marked in dark gray) which have been allocated an operation by the control generator.

When the first instruction, AND R3, R1, #-4, enters the control generator, that instruction looks up each source operand in the current producer table. Since R1 has no current producer, it is added to the list of live-ins. No other nodes in the subgraph create results that this operation consumes, so the AND instruction can be assigned to node A in the first row of the CCA. The current producer table is updated to reflect that R3 is generated by CCA node A, and R3 is marked as potentially live-out. The opcode AND and constant -4 are stored as the function executed by node A. The state after processing the AND instruction is reflected as Step 1 in Figure 4.9.

Spill code for R3 is the next instruction entering the control generator. The compiler guarantees that any spill code within the subgraph is only for transient values, and thus can be optimized away without affecting the correctness of the program. In this example the spill code stores R3 to stack offset 20. Since R3 is produced by node A, that node value is stored with an index of the stack offset in the CAM. Future instructions that use values spilled on the stack, use the CAM to determine which node in the CCA generates the instruction's inputs. Step 2 in Figure 4.9 shows the control configuration state after mapping the store instruction.

Following the spill instruction, the SEXT instruction enters the control generator. Since this instruction uses R2, and R2 has no producer in the current producer table, R2 is marked as live-in and the instruction is placed at node B in the first row of the CCA. This instruction produces a value for the spilled register R3, so the current producer of R3 is changed to node B, and the live-out bit of R3 remains set. When the next AND instruction is mapped,

a look up of its source operand R3 shows that node B produces it. This means that the AND operation must be placed in the row below node B, in this case node G. The current producer table is then updated to reflect that R2 is now produced by node G.

The next retired instruction is the spill code load for R3. The control generator looks up the spill offset in the CAM and finds that node A generated the value being loaded. Thus, the LD instruction resets the current producer of R3 to node A, and it remains marked as live-out. After the spill code load, an OR instruction with sources R2 and R3 is processed. Both of these sources are produced by other nodes in the subgraph. Since it is dependent on node G, this operation must execute in the third row. It is placed at node K and updates the current producer table accordingly. In addition, because the operation requires a source from the first row (R3), a move must be inserted in row 2. Moves are necessary because only adjacent rows in this CCA architecture are directly interconnected. This move is marked in light gray at the bottom of step 6. Similar to previous instructions, the AND is inserted in the last row of the CCA. The final instruction, an RET, marks the end of this pattern.

Once the end of the subroutine is reached, the control data is not yet ready to be written to the BTAC and CCA configuration cache, since there exist more live-outs than are supported by the execution system. The compiler is responsible for proving that only a limited number of live-outs exist in each pattern. Therefore, to determine which ones are not actually live-out, it is necessary to monitor the retiring instruction stream and unset the live-out bit for any register that is defined before used.

Determining true live-outs can either be done by waiting for other instructions to naturally kill potential live-outs, or by having the compiler insert artificial instructions to ensure that false live-outs are killed quickly. Regardless of the strategy, the latency of killing live-outs should prove irrelevant to system speedup as prior work [45] has shown that moderate latencies are likely between trace retirement and recurrence.

If at any point the control generator cannot map a subgraph onto the underlying CCA execution unit, then it simply aborts control generation for that pattern. This allows applications compiled for CCA subsystem 1 to run on CCA subsystem 2 even when the second may not support all the subgraphs that the first supports. Providing the dynamic control

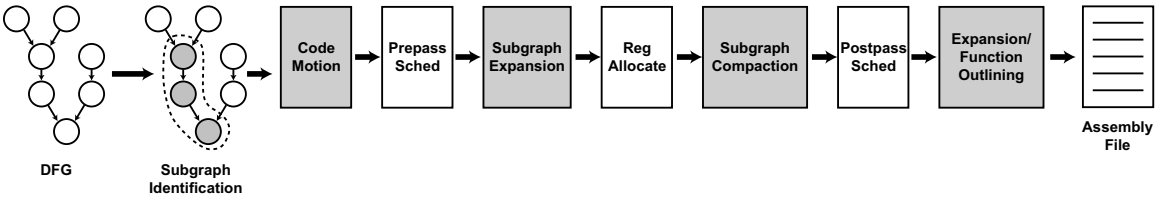


Figure 4.10: Compiler flow diagram. New steps in the compilation process are shown in gray.

generator as part of the CCA subsystem is key to the retargetability of the system.

The specific hardware structure and overhead of control generators is discussed in Section 4.12.

4.7 Compiler Code Generation

In order to exploit the specialized CCA hardware, a CCA cognizant compiler requires several new steps in the code generation process. The overall structure of the compiler flow is shown in Figure 4.10; steps added for CCA compilation are gray in this figure. Normal code compilation has three major steps: scheduling, register allocation, and postpass scheduling of spill code. At the beginning of compilation, a CCA compiler must determine which dataflow subgraphs should execute on the CCA. The remaining complexity of compiling for a CCA stems from the fact that some phases of compilation need to treat the subgraphs as atomic units and other phases need to understand each constituent node of the subgraph. Each of these phases is explained in detail in the remainder of this section.

4.7.1 CCA Compiler flow

Subgraph Identification: Given a dataflow graph as input, subgraph identification determines which portions should be executed on the CCA. This is very similar to the problem of technology mapping in VLSI design. In the general case, where the subgraphs are not necessarily trees, the problem is NP-complete [3]. Difficulty of the problem is the primary reason subgraph identification is performed at compile time instead of runtime.

Heuristics for solving subgraph identification have been the subject of much related work [3, 80, 87]. This is a very complicated issue, discussed further in Chapter 5.

From a high level, subgraph identification is performed in two steps. First, subgraphs are enumerated within a basic block or superblock, using a branch and bound algorithm. This algorithm generates the set of all subgraphs capable of being executed on the target CCA. In the case that a block is too large for full enumeration, the block is intelligently split into smaller pieces, each of which is fully enumerated.

After enumeration, the second step of subgraph identification is selecting which of the enumerated subgraphs to execute on the CCA. At issue is that each operation in the dataflow graph may appear in multiple subgraphs, yet each operation can only be mapped onto the CCA as a member of one subgraph. Thus, it is necessary to either replicate operations or a subset of subgraphs must be selected to maximize performance subject to the constraint that each operation appear in only one subgraph. Beyond that, it is also necessary to determine if the target CCA is capable of executing the subgraph more efficiently than the constituent operations on the baseline processor. For example, if a subgraph consists of two dependent ADD operations, and the latency of the target CCA is three cycles, then executing that subgraph on the CCA is not worth the overhead. In this work, subgraph selection is accomplished using a dynamic programming heuristic described in [32].

It is important note that subgraph identification is performed before register allocation. Performing subgraph identification after register allocation introduces many false dependencies within the dataflow graph, and hinders the size of the subgraph that can be discovered. Indirect evidence of these dependencies exists in the effectiveness of register renaming logic in superscalar processors. Even though false dependencies are a major problem, most related work performed subgraph identification after register allocation, so it could be done at link-time or run-time.

Code Motion: After subgraph identification, the selected subgraphs are collapsed into a single instruction. In order to effectively mark subgraphs as a special procedure calls for the hardware, it is essential that the scheduler maintain the instruction order such that the subgraphs appear contiguously in the code. Collapsing the subgraph into a single node cleanly prevents operation reordering without altering the scheduler internals.

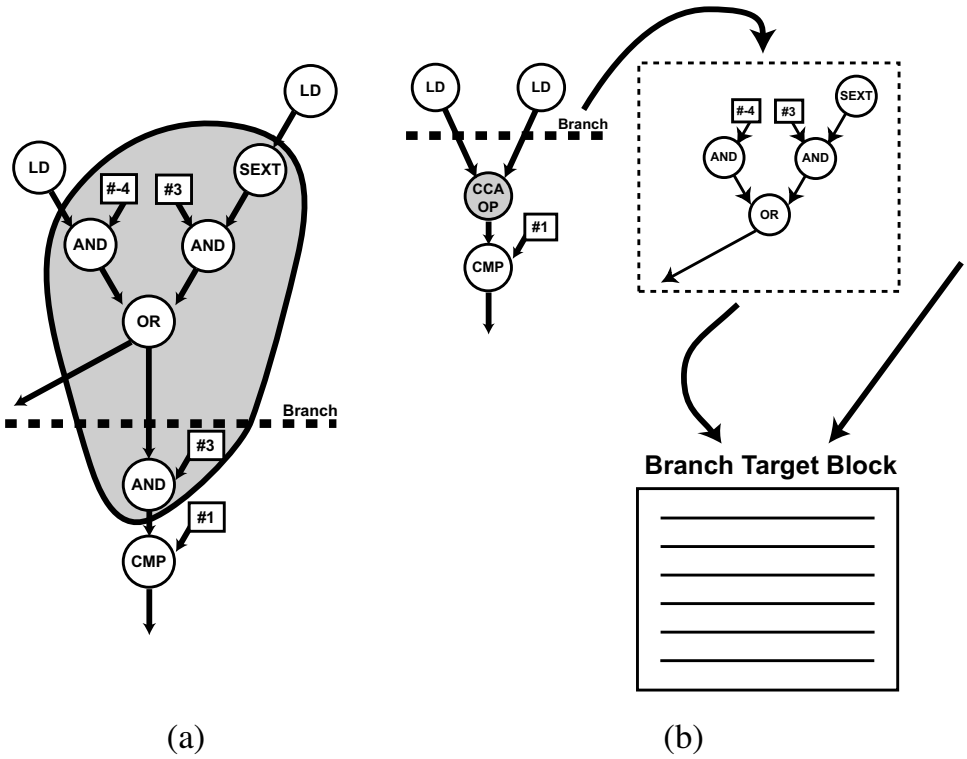


Figure 4.11: The process of downward code motion as (a) the cross branch subgraph is identified and (b) code is replicated in a new block

When collapsing the subgraph, a problem arises if the subgraph crosses branch boundaries. Previous work has shown that preventing subgraphs from crossing branch boundaries greatly constrains the size of the subgraphs [131]. Thus, a decision must be made as to where to insert the CCA node relative to the crossed branch. We consider the two extreme possibilities: before the first branch and after the last branch. Both choices have ramifications which must be corrected in the code. The process of placing a CCA operation after the last branch boundary is termed *downward code motion* and placing the CCA operation before the first branch boundary is termed *upward code motion*. Without loss of generality, each form of code motion is considered for a single branch operation.

In downward code motion, the subgraph is assumed to span the not taken direction of the branch. The problem that arises is there could potentially be portions of the collapsed subgraph which need to be executed before the code at the branch target is executed. Consider the example in Figure 4.11(a), which is a portion of the dataflow graph from the Ri-

jndael encryption benchmark. The subgraph identified for collapsing is encircled in gray. If the collapsed node is executed after the branch boundary, the application will execute correctly as long as the branch is not taken. However, if the branch is taken, then there are operations within the collapsed subgraph that did not execute but should have. These are the operations from Figure 4.11(a) that are within the encircled gray subgraph and above the dotted branch line.

After placing the collapsed subgraph below the branch boundary, the portion above the branch must be replicated. Figure 4.11(b) shows this process when the branch target is a block of code with a multiple entries. A new block is created with the code region from the collapsed node. This code region then unconditionally branches to the original branch target. In the case where the target block has only one control flow entry point, this new block is simply collapsed into the beginning of the branch target block. This process is essentially the same as the bookkeeping code induced through downward code motion used during trace scheduling [44].

Downward code motion easily extends to patterns which cross multiple branch boundaries. Generally speaking, executing subgraphs that cross branch boundaries increases the size of the computation subgraphs executed on the CCA, which improves performance. The trade off is increased code size from operation replication.

The alternative to downward code motion is to place the collapsed subgraph above the branch boundary, or upward code motion. In this case, the CCA could potentially execute code that should never have been executed, and therefore speculates that the branch will not be taken. If the branch is taken, then code must be inserted to repair the incorrectly executed instructions. Additionally, operations that could potentially cause exceptions, such as a divide or load operations, must not be speculatively executed to guarantee correct execution.

The CCA compiler system implemented in this work exclusively uses the downward code motion process, placing the CCA operation after the branch. This method always produces functionally correct code regardless of excepting instructions.

One potential area where downward code motion has difficulty is if a value produced by a CCA instruction is consumed by the branch. For example, in Figure 4.11(a), if the live

out from the OR operation was used to determine whether or not the branch is taken, then this subgraph cannot be moved below the branch. In this case, the CCA compiler rejects this potential subgraph as a target for collapsing.

Prepass/Postpass Scheduling: These two phases of compilation are unchanged from the standard compiler. Later in compilation, the subgraphs are turned into special function calls using the BRL' instructions, and thus, it is important to keep all of the subgraph instructions contiguous in the schedule. This is the main reason why subgraphs are compressed into atomic instructions.

Subgraph Expansion: While scheduling considers the subgraphs as atomic units, register allocation needs to consider each instruction separately in order to properly assign the registers to the internal values. Recall that processors without CCA subsystems must still be able to execute the code generated for processors with CCAs. This mandates that the subgraph must be register allocated. Without expanding the subgraphs, it is difficult for the register allocator to correctly construct live ranges and assign registers.

Register Allocation: Expanding the subgraphs before the register allocation allows this phase of compilation to be relatively unchanged. Registers are simultaneously assigned to all instructions, including the expanded subgraphs, just as they would normally be. The only change has to do with the addition of some caller save code for the subgraph. Recall that the subgraph will be implemented as a subroutine call using the BRL' instruction. The BRL' will overwrite the link register, if it is not saved to the stack. Thus, a save and restore of the link register are added on either side of the subgraph. No additional caller save code is necessary, since we know exactly which registers will be used in the subgraph and have already allocated appropriately. In calling conventions where the link register is already callee saved in the function prologue, this additional code is not necessary.

An optional optimization to register allocation is to intelligently prioritize the variables to be allocated. Since the CCA control generator is capable of collapsing spill code of transient values within subgraphs, there is no need to allocate a register for those values at the expense of other variables. Giving these transient values very low priorities, guarantees that register allocation will spill them if necessary, and effectively increases the number of registers available to the machine.

Subgraph Compaction: After register allocation, the full subgraph is again compressed to an atomic node in preparation for postpass scheduling. This process is complicated slightly by spill code that is introduced in relation to the subgraph. If a transient value in the subgraph is spilled, e.g. R3 in Figure 4.9, then this must be combined into the subgraph. By placing this spill code in the subgraph, the compiler guarantees that the results are not needed outside of the subgraph and these loads/stores can be optimized away by the CCA subsystem. If a value that is live-out of the subgraph is spilled and also consumed in the subgraph, then the store that spills the live out is replicated outside of the subgraph. A copy of the store must remain in the subgraph so that the control generator can determine which node produced the spill value. Once the subgraphs are compacted, postpass scheduling is performed.

Function Outlining: After postpass scheduling, the subgraphs are again expanded into their constituent nodes. Each subgraph is moved to a separate portion of the code and a BRL' is inserted at the former location of the subgraph. This process is referred to as *function outlining*.

The technique of function outlining (sometimes called procedure abstraction) has been used in previous work [71, 79] for code size reduction. Since groups of instructions are often repeated at several different places within an application, function outlining can combine these instances into one procedure. While the primary purpose of our function outlining is to delineate subgraphs for the hardware, it also provides us with code compression to help offset some of the code replication from subgraphs that cross branch boundaries. It should also be noted that the code size reduction could be improved by making the register allocator more proactive in assigning the same register values to isomorphic subgraphs.

With function outlining complete, an assembly file is output that can be run on any processor which recognizes the BRL' instruction.

4.8 Architecture Framework Experiments

Our experimental system was built on top of the Trimaran compiler infrastructure [121]. Trimaran was retargeted for the ARM instruction set and augmented with a parameterized

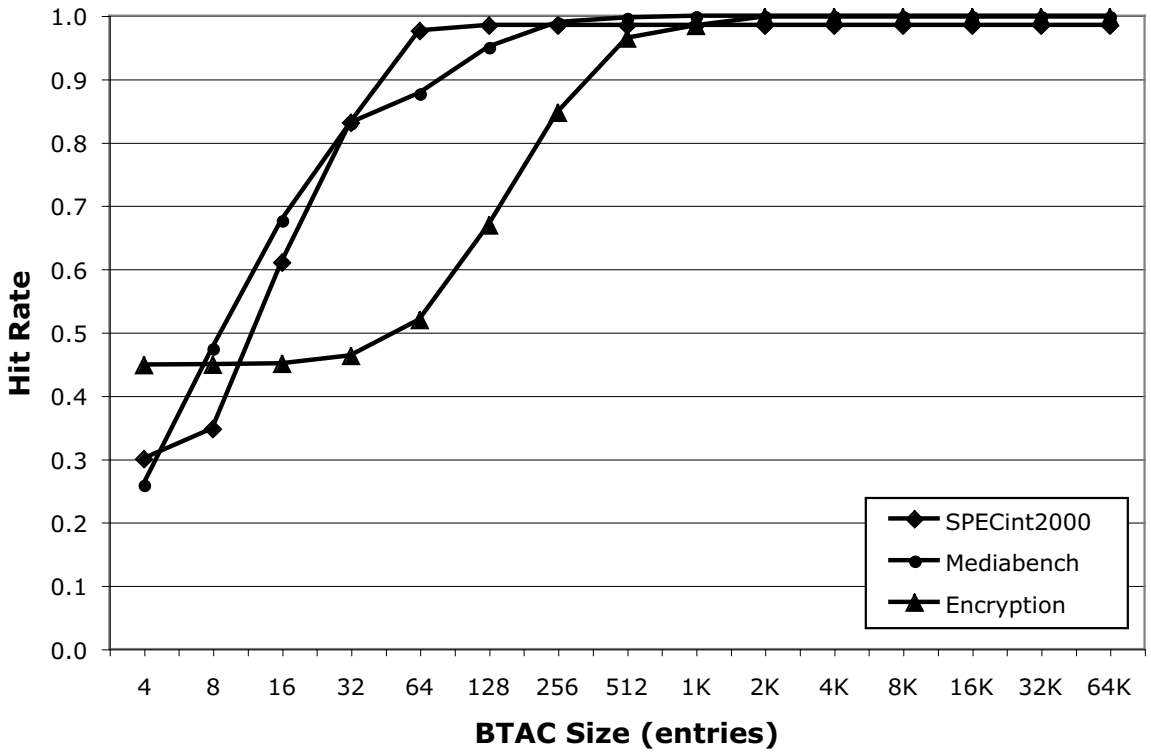


Figure 4.12: BTAC hit rate with various entry sizes

subgraph matcher to recognize dataflow subgraphs that map onto the underlying CCA infrastructure. Once the subgraphs are identified, code motion, scheduling, and the rest of the steps described in Section 4.7 are performed. For evaluation, SimpleScalar [9] ARM was modified to implement the CCA interface and configured to match the ARM-926EJ [5]. The ARM-926EJ is a fairly simple, in-order, five-stage pipelined processor with 16K, 64-way associative instruction and data caches.

For our experiments, we evaluated a set of embedded and general-purpose benchmarks consisting of five encryption related applications (Blowfish, MD5, RC4, Rijndael, and SHA), and a subset of the MediaBench [75] and SPECint2000 applications. The range of our application set was limited by the current capabilities of the ARM port of the Trimaran compiler suite.

BTAC Size Study: Before evaluating the effectiveness of the CCA, we investigated several possible configurations for the BTAC, which holds the branch addresses and live-in

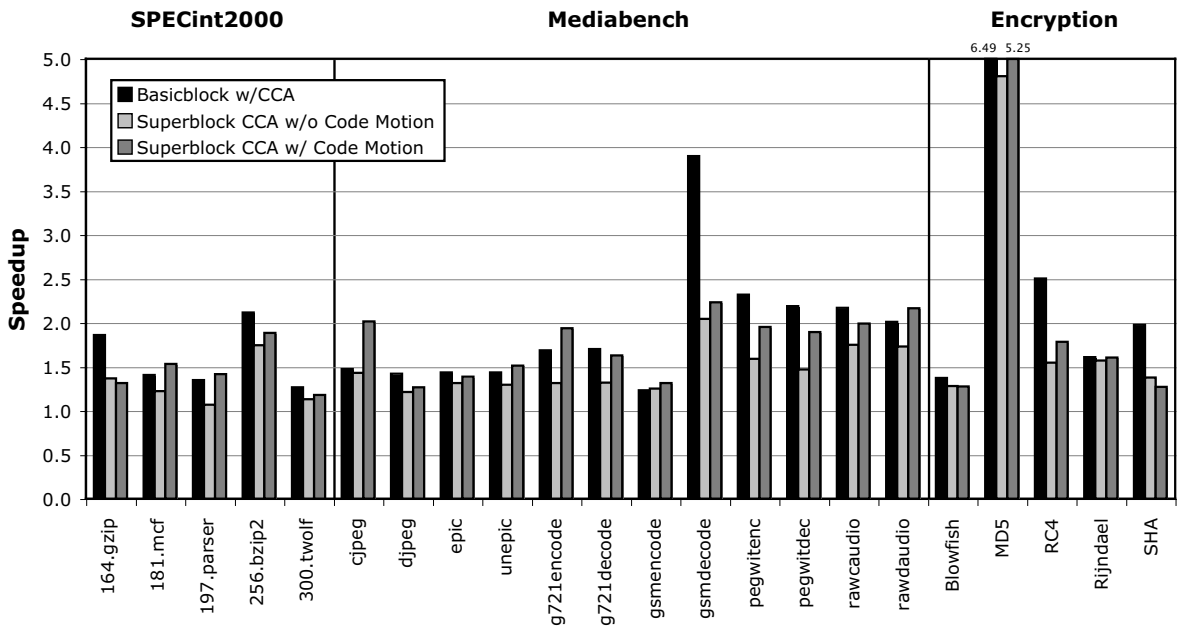


Figure 4.13: Speedup of basic block and superblock code when executing with a general purpose CCA

information for the CCA subsystem. Figure 4.12 shows the BTAC hit rate given several different BTAC sizes. The three lines indicate the average hit rates of the BTAC for the encryption, MediaBench, and SPECint2000 applications. Interestingly, even with only 4 entries, the BTAC was able to capture a fairly large number of the marked subgraphs. For example, in the encryption domain, 45% of the subgraphs were captured. In the remaining experiments, we used to use a 512 entry, four-way associative BTAC, which achieved a hit rate average of 98.5% across all benchmarks.

Performance Study: Figure 4.13 shows the relative speedups that were achieved for code compiled using both basic blocks and superblocks [89]. For each benchmark, three bars are shown. The first bar is the speedup of basic block code with a CCA relative to basic block code compiled without CCA subgraphs. Both of the next two bars are superblock code with a CCA relative to superblock code compiled without CCA subgraphs. The first of the two superblock bars is for code without code motion applied, which limits the subgraphs by not allowing them to cross branch boundaries. The second superblock bar was generated

by allowing the compiler to perform code motion.

All of the results in Figure 4.13 used the general purpose CCA designed in our previous work [28] and shown in Figure 4.8. Synthesis results showed that this CCA used 0.61 mm^2 of die area, and gave average speedups for the basic block code of 1.60 for SPECint2000, 1.91 for MediaBench, and 2.79 for encryption applications. The encryption applications showed the most improvement because they tend to have the largest amount of computation between memory accesses, thereby creating larger subgraphs to map onto the CCA. The results show that substantial performance gains across a wide range of applications are realized with a relatively inexpensive compute accelerator that is tightly integrated into a processor. The CCA provides a more efficient hardware substrate to execute the subgraphs, which translates into performance gain.

One trend to note in this graph is that in many cases, using superblock code had a smaller relative speedup than basic block code. Intuitively, superblock code should result in the identification of larger patterns, which should directly translate into improved performance over the basic block code. However, register pressure is an important performance issue in the ARM processor. Forming superblocks caused an increased size in register live ranges. This increase in live range size dramatically affected the amount of register spill code which the compiler was unable to optimize using the CCA.

Applying the code motion techniques discussed in 4.7 to the superblock code resulted in improved performance in most cases since adding the code motion optimization allowed the compiler to find patterns which cross branch boundaries. In some cases, such as cjpeg and g721encode, the performance improvement was as much as 50%, while a few other cases suffered slight performance degradation. This performance degradation is a result of code motion enlarging register live ranges as operations are pushed down below branches and new code is inserted in the target blocks. Again this increases register pressure and may lead to increased spill code.

Custom CCA Designs: Though a general purpose CCA design provides impressive performance gains across a diverse set of applications, tailoring a CCA to either a single application or a domain of applications can yield a more area-efficient design. In order to explore the design of application and domain specific CCAs, the compilation process

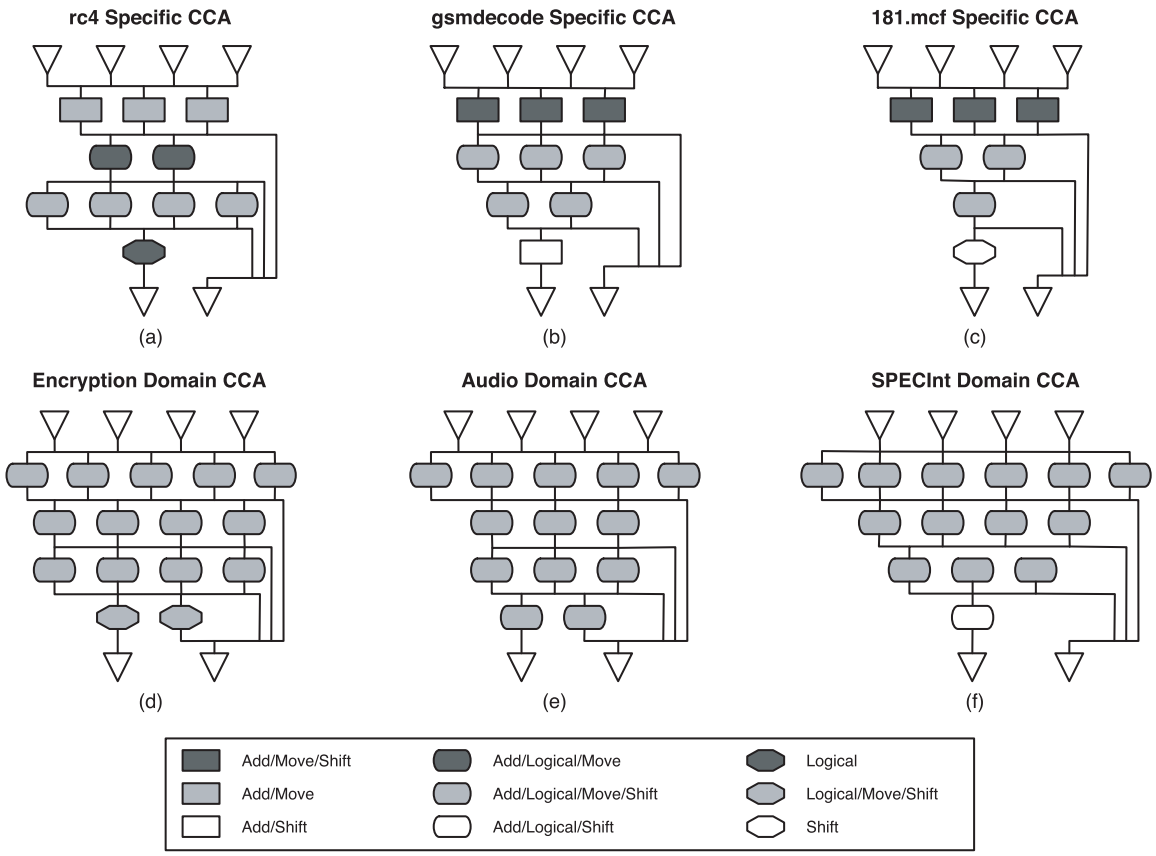


Figure 4.14: Application specific and domain specific CCA design results

was augmented so that when subgraphs are identified, the operations which comprise the subgraph and their profile weights are passed to a scheduler. The scheduler then incrementally builds a reservation table for each subgraph. After all subgraphs in the application have been identified, the scheduler then builds the application-specific CCA structure as the union of all of the necessary reservations for each subgraph meeting a minimal profile weight requirement. Lastly, domain specific CCA structures are built as the union of all application specific CCAs synthesized for a particular domain. This approach is not intended to produce optimal CCAs, but rather illustrate the flexibility of the proposed architectural framework to support a wide variety of CCA designs.

Figure 4.14 demonstrates the structure of a set of automatically generated application and domain specific CCAs. The top row of Figure 4.14 consists of one application specific

Description	Design	Control	Delay	Cell area
Application specific CCA for RC4	Figure 4.14(a)	73 bits	4.10 ns	$0.25\ mm^2$
Application specific CCA for gsmdecode	Figure 4.14(b)	84 bits	6.04 ns	$0.33\ mm^2$
Application specific CCA for 181.mcf	Figure 4.14(c)	55 bits	5.68 ns	$0.26\ mm^2$
Domain specific CCA for encryption	Figure 4.14(d)	181 bits	5.69 ns	$0.45\ mm^2$
Domain specific CCA for audio	Figure 4.14(e)	140 bits	5.86 ns	$0.46\ mm^2$
Domain specific CCA for SPECint	Figure 4.14(f)	171 bits	6.05 ns	$0.56\ mm^2$
General purpose CCA from [28]	Figure 4.8	172 bits	3.19 ns	$0.61\ mm^2$

Table 4.1: Synthesis results for various CCA designs

CCA designed for an application in each of the presented domains, encryption, audio, and SPECint, respectively, while the bottom row consists of the set of domain specific CCAs.

Table 4.1 presents an analysis of the design costs for each of the CCAs shown in Figure 4.14. The table includes the number of control bits necessary to configure the CCA, the delay through the CCA, and the area of the CCA. Each of these designs was synthesized with Synopsys design tools using a $130nm$ Artisan library. In order to provide insight into the cost of adding a CCA to an actual ARM core, we note that the actual area of an ARM-926EJ is $5.0\ mm^2$. Also important to note is that the design for the general purpose CCA from [28] was hand-tuned to minimize the number of levels including adders in the CCA thus significantly reducing delay through the CCA. A more intelligent automated design process for our application and domain specific CCAs would likely provide improvements in terms of both area and delay.

Figure 4.15 demonstrates the performance improvements offered by the designs shown in Figure 4.14. In this graph, the first bar indicates the performance of the general purpose CCA relative to the baseline processor with no CCA. The second bar demonstrates the speedup achieved by using the domain specific CCA designed for the domain that the application belongs to, assuming a 1-cycle delay through the CCA. The third bar demonstrates the performance of the same CCA as the second, but assumes a 2-cycle delay through the CCA. The fourth bar shows the speedup of using the application specific design shown in Figure 4.14 for each application in the same domain. This means that for applications within the SPECint domain, all application specific speedup is calculated using the CCA

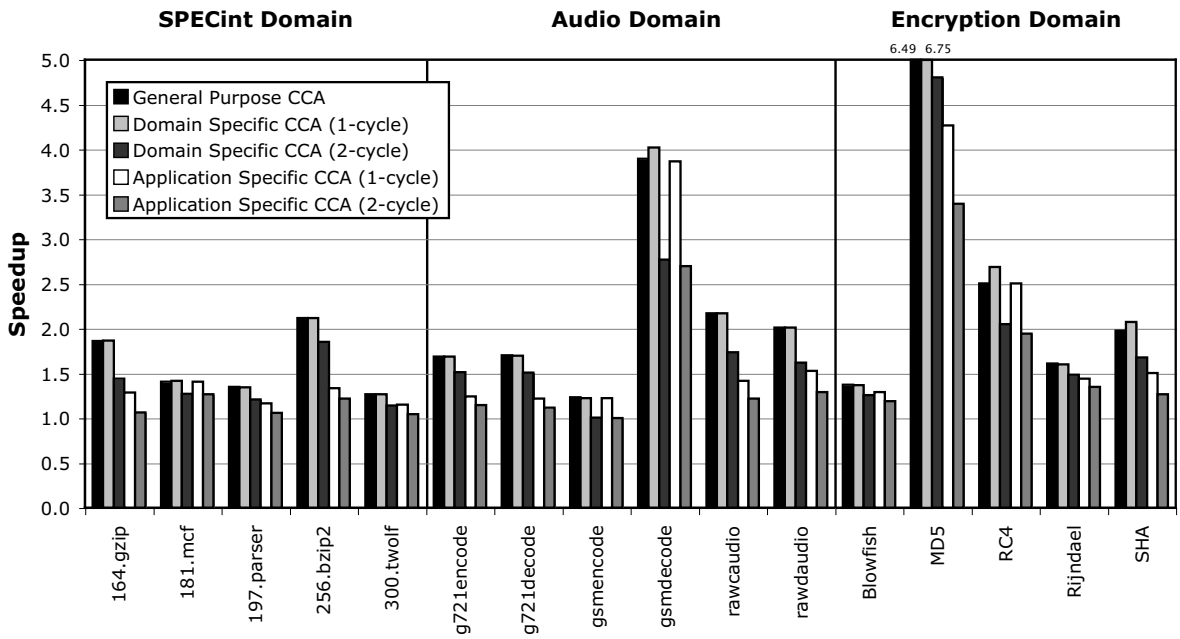


Figure 4.15: Application specific and domain specific speedup. For the SPECint domain, application specific speedups are generated using the CCA designed for 181.mcf, for the audio domain using the design for gsmdecode, and for encrypt domain using the design for RC4.

designed for 181.mcf, for the audio domain using the CCA designed for gsmdecode, and for the encryption domain using the CCA designed for rc4. The decision to use the application specific design for a variety of different benchmarks was to show the applicability of these designs across a set similar benchmarks. The last bar utilizes the same CCA structure as the fourth, but assumes a 2-cycle delay through the CCA.

From Figure 4.15, it is clear that a domain specific CCA design can closely match the performance of the general purpose design at lower cost, provided that it can fit into the 1-cycle delay constraint. Further, the application specific CCA designs tend to closely track the performance of their respective domain specific designs while still proving beneficial to a variety of other applications within their domain at nearly half the area overhead. It is important to note that the domain specific designs tend to provide marginal performance gains over their application specific counter parts due to their ability to catch the few subgraphs that had been pruned from the application specific CCA design.

4.9 Architecture Framework Summary

In the previous sections, we presented the design and implementation of a flexible architectural framework for supporting *transparent instruction set customization* using *configurable compute accelerators*. The use of this framework reduces both system design and verification costs. A general purpose core implementing the pre-defined CCA interface need only be designed and verified once. The core may then be augmented with several different styles of compute accelerators offering a wide range of systems with performance characteristics tailored to an application or domain of applications. In addition to the architecture framework, we also demonstrate the compilation process used to target an application toward a particular CCA architecture.

Synthesis results demonstrate the feasibility of the proposed architecture framework in terms of meeting the timing and area constraints of common embedded processors. Further, experimental results demonstrate average performance gains of 2.21x for domain specific CCA designs, with modest cost overhead beyond the original processor design. The range of applicability of these designs may be restricted or expanded in order to both meet area constraints and satisfy performance goals for a specified range of applications. The proposed architectural framework provides system designers with a low-cost solution for designing a wide variety of high-performance systems by augmenting a single core with multiple implementations of an accelerator subsystem.

4.10 Control Generation for Dynamic Accelerator Targeting

The previous sections of this chapter discussed the system architecture and software side of a statically identified - dynamically realized transparent instruction set customization framework. In the remainder of this chapter we focus on the last step: efficiently performing run-time control generation in hardware. We show that it is indeed possible to generate the control quickly with low overhead for a divergent set of accelerators using a single, abstract control generation model. To demonstrate this model, we give imple-

mentation details showing how to generate control for the varied computation accelerators previously proposed: a sparsely-connected array of combinational logic elements (CCA), and a lookup-table based accelerator (PCFU).

The contributions of these sections are threefold:

- We describe a generalized framework for dynamic control generators, and show how this framework can target varied types of computation accelerators.
- We provide characterizations of the hardware properties of these control generators, showing that they have limited overhead over a baseline processor.
- We introduce novel algorithms for control generation targeting sparse arrays of combinational logic, and LUT-based accelerators.

4.11 Dynamically Mapping Architectural to Microarchitectural Instructions

The objective of control generation is to dynamically create complex instructions by mapping subsets of an application onto a set of hardware accelerators. In essence, this can be thought of as the inverse to the micro-operation generation that is performed on Pentium processors, wherein CISC instructions are broken down into RISC-like micro-operations for execution on the hardware. In contrast to micro-op generation, CISC-on-demand is only applied selectively to those subgraphs that can be executed on the hardware accelerators provided in the processor implementation. To accomplish dynamic control generation, the pipeline needs to be extended to support subgraph identification and CISC operation generation. Issues related to this process are described in the remainder of this section in the context of a generalized accelerator. Section 4.12 describes detailed control generator implementations of two very different accelerator classes.

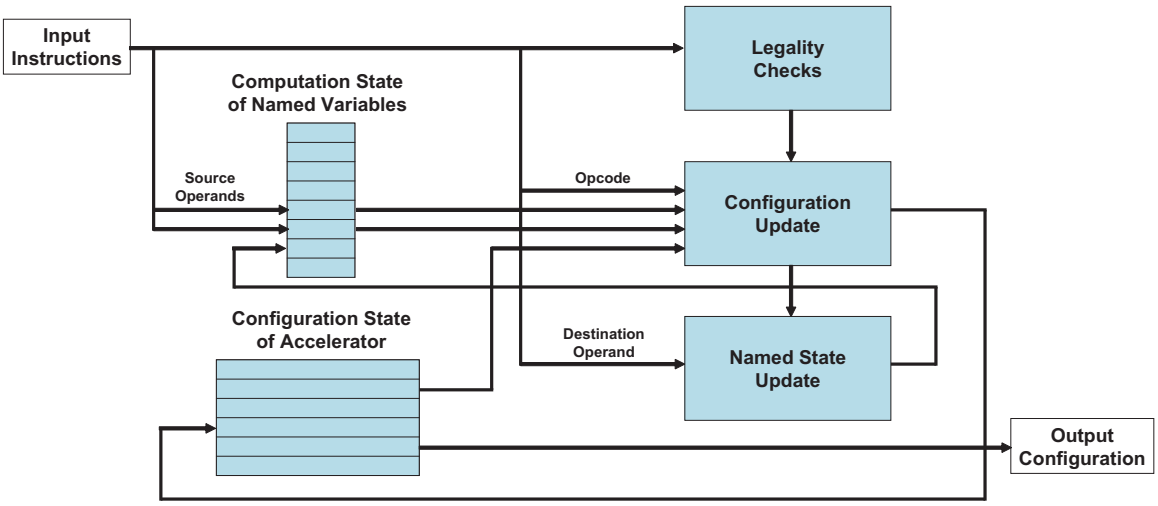


Figure 4.16: Structure of a control generator

4.11.1 Structure of a Control Generator

The cornerstone of transparently mapping instructions to accelerator subsystems is the ability to generate control for the accelerators using instructions from the baseline instruction set. In essence, control generators *translate computation specified using the baseline instruction set into an equivalent computation to be understood by the accelerator*. Since all hardware-based control generators perform this task, they share many common properties. This section describes those general properties, before several accelerator-specific control generator designs are described in Section 4.12.

Figure 4.16 shows the structure of an accelerator control generator. Instructions enter the control generator after retiring from the baseline pipeline. The instructions undergo some legality checks to make sure that they are capable of being executed on the targeted accelerator, and in parallel access some state based on their source operands. If the operation is legal, then the state from the source operands, the current configuration state, and the instruction itself are used to generate an updated configuration. Updates to the configuration are used to modify the named state associated with the destination operands. Once the entire computation is mapped instruction-by-instruction, this configuration will be sent to the control cache, enabling the subgraph to execute on an accelerator. Details on the

purpose and general structure of these components are described in the remainder of this section.

Configuration State: The configuration state of a control generator essentially stores the control, i.e., a description of the dataflow subgraph to be executed in a microcode language the accelerator understands. As instructions enter the control generator, the configuration state is updated to reflect the computation of the subgraph that has been processed up to that point.

The data stored within the configuration state is highly dependent on the structure of the accelerator being targeted. For example, configuration state might include lookup-table (LUT) values in an FPGA style accelerator, MUX select values in an accelerator that requires routing data to various computation units, or even full instructions if the accelerator is instruction driven, such as SIMD engines.

Computation State for Named Variables: In addition to the current configuration state of the accelerator, state is needed to correspond to named variables within the instructions. In most cases, named variables refer to registers specified in the incoming instructions. Intuitively, it makes sense that some state is required per register to generate control for a computation, since registers provide the state to communicate data between instructions in the baseline pipeline. Similar to the base pipeline, this named variable state is used to determine the relationship and communication of different instructions within the dataflow subgraph to be accelerated.

As previously mentioned, the named variables typically referred to in computation are register numbers. This is not necessarily the case, though. Named variables can also include variables stored at any memory location that is constant within a subgraph invocation. For example, consider an instruction set that places spilled variables on the stack using stack-pointer-plus-displacement memory operations. If the stack pointer only changes at function entry and exit (a common ABI convention), then the control generator can guarantee every load or store to a particular stack offset corresponds to the same variable. Using this knowledge, the control generator can recognize limited forms of communication through memory, which allows it to take advantage of any additional communication bandwidth the accelerator may have over the baseline pipeline.

Legality Checks: Legality checks simply determine if the incoming instruction can be mapped onto the targeted accelerator. Since one binary can be used with any system, subgraphs identified in the binary can may not be executable on this particular accelerator. The legality checks ensures that the control is not generated if a particular subgraph is not executable on its target accelerator.

Checking for functionality is a good example of a legality check. Suppose a control generator was trying to map a subgraph with a divide instruction, but this particular accelerator did not have a divider. The legality check would ensure that no control was generated for this subgraph, leaving the subgraph to execute on the baseline pipeline.

Configuration Update: Configuration update simply refers to the logic used to update the configuration state of the control generator. This is the part of the control generator that does the translation from instructions in the baseline instruction set to one or more instructions fed to the accelerator. Configuration update produces a new accelerator configuration based on the current configuration, the computation performed by the incoming instruction (i.e., its opcode), and the relationship that this instruction has with the previous computation (i.e., state from named variables). If at any point the update logic cannot translate an instruction, control generation for this subgraph is aborted, and nothing is written to the control cache.

As would be expected, the function of the configuration update logic varies a great deal with the structure of the targeted accelerator. For example, if targeting an array of computation units, the configuration update would need to identify which node in the array to map the incoming instruction onto, and set routing control values appropriately. If the update logic was targeting an FPGA, it would need to generate new LUT entries for that computation substrate.

It is important to keep in mind that mapping computation onto many accelerator styles (e.g., placement and routing in FPGAs) is a computationally demanding task. We are not proposing to push this burden onto the configuration update logic. It is the responsibility of the compiler that identifies the subgraphs to express that computation in a way that enables the configuration logic to remain relatively simple.

Named State Update: After the updated configuration state has been generated, it

is necessary to compute the new state for the named variables. This named state update logic generates information about the computation used to create the destinations of the incoming instruction. That is, given that the input instruction has been mapped onto the accelerator, this logic describes what information needs to be known about the destination register to continue mapping. Using the example of an array shaped accelerator again, this information might include which node of the array the destination can be found at. State keeping track of where data values are produced enables future instructions to route that data to the inputs of their respective computations (this is explained in more detail in Section 4.12).

Optionally, the named state update unit can also perform a type of register renaming. For example, if the control generator was targeting an accelerator that had 32 internal registers, but the baseline instruction set only had 16 addressable registers, the named state update logic could perform renaming to take advantage of the additional internal state, despite the limitation of describing computation using only 16 registers in the baseline instruction set.

Once the end of a subgraph is identified, the initial PC of the subgraph and configuration state of the control generator is sent to the control cache ensuring that subsequent encounters of the BRL-to-subgraph will be executed on the accelerator.

4.12 Implementation of Control Generators

To more firmly illustrate control generation techniques, this section details control generators for two styles of previously proposed hardware accelerators: (1) an array of combinational logic units proposed in [28], and (2) a lookup-table based accelerator introduced in [129]. These accelerator styles were chosen because they support a wide range of computational patterns (often a superset of previous work), and they represent very different substrates for execution (e.g., LUT vs. fixed logic based). These factors make it likely that the control generation techniques presented here are widely generalizable to many other types of accelerators.

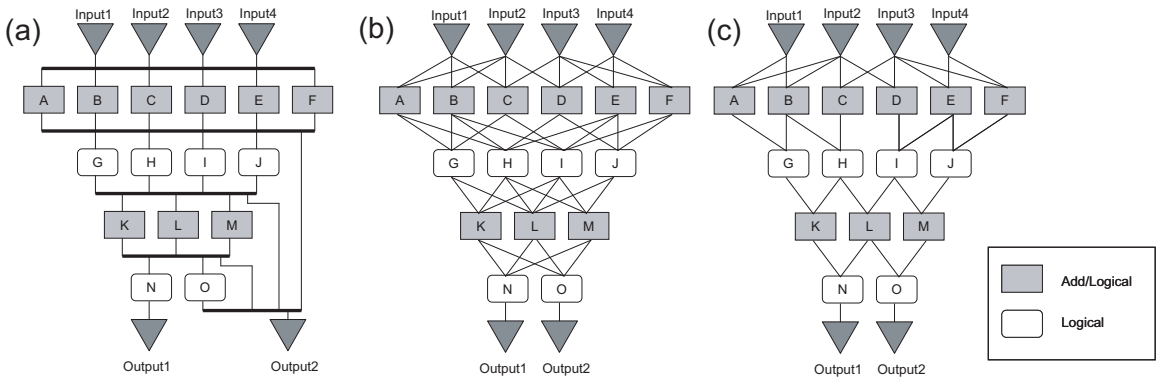


Figure 4.17: Combinational logic arrays (a) with a full cross bar between rows, (b) with moderate interconnect, and (c) with sparse interconnect

4.12.1 Arrays of Combinational Logic

When designing hardware to execute dataflow subgraphs, arrays of combinational logic are a fairly intuitive solution. The height of the array allows for compression of operations and reduces the need to write transient values back to the register file, while the width of the array allows for exploitation of natural parallelism that exists in the computation. Using combinational logic as the building block also allows one to exploit highly optimized macrocells, as designing circuits for these functions has been the subject of decades of research.

Based on these observations, previous work [28] analyzed the critical computation subgraphs in many common applications and proposed the combinational logic array in Figure 4.17 (a). This array has four inputs, two outputs, and four rows of logic. Each row has a number of nodes that can execute operations, for example, the first row can execute ADD, SUB, AND, and several other simple logic operations. The second row in Figure 4.17 (a) can only execute bit-wise logic operations. In this design, each of the rows is connected with a full crossbar. This design was found to execute more than 80% of the critical subgraphs across a wide range of applications.

While this design was shown to be effective from a software perspective, the crossbar between rows of the array is a substantial problem. The full interconnect has several detri-

mental effects. First, it implies several long wires in the design, which increases latency through the array dramatically. As technology scaling continues, the affects of long wires will only be exacerbated, meaning that combinational arrays must be restricted to local communication in order to remain feasible. Second, each input writing to the interconnect must drive a much higher capacitance. Third, high fan-in multiplexers are needed at the input of each node. All of these factors either increase latency of the combinational array or increase the die area and power consumption.

From a control generation perspective, the full crossbar is much easier to handle than an incomplete interconnect. With a full crossbar in the accelerator, determining which node of the array to map an instruction onto is simply a matter of determining the correct row, since all nodes within a row are logically equivalent. A high-level algorithm for control generation with a full crossbar is illustrated in [30]. We extend that algorithm here, providing more low level details, and enabling it to work without full crossbars between rows.

In this chapter, two new combinational logic arrays were designed, as shown in Figures 4.17 (b) and (c). The purpose of these designs is not so much to propose new accelerator designs, as it is to demonstrate control generation strategies that are effective on array based accelerators with sparse interconnect. Note that these designs support a superset of the subgraphs targeted by some previous work [17, 102, 111] using similar execution substrates, meaning that the control generation techniques presented here are applicable for those accelerators as well.

4.12.2 Control Generation for Sparse Arrays of Combinational Logic

Generating control for sparse arrays of combinational logic is a challenging problem. Essentially, the goal is to map the dataflow graph identified for acceleration onto the computation array such that data values flow correctly from operation to operation. This can be viewed as a simpler form of a place-and-route problem.

The control generator described here uses the template in Figure 4.16. For this type of accelerator, the configuration state includes the operations performed at each node in the array and the MUX values which select the inputs for each node. Computation state for

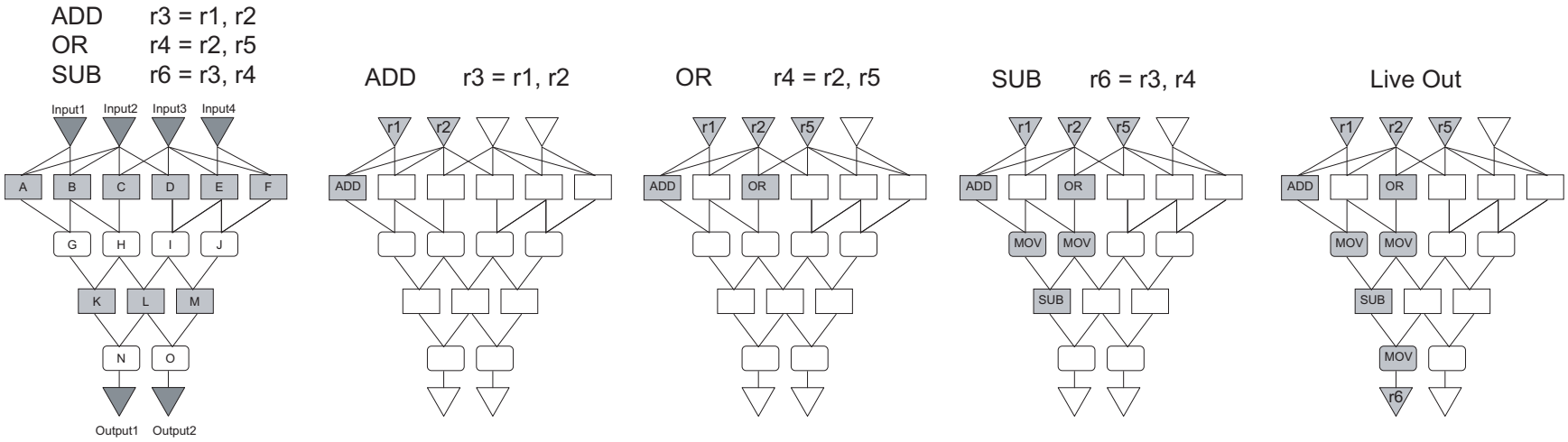


Figure 4.18: Example of control generation for a sparse array of combinational logic.

named variables is a table with an entry for each named variable stating which node in the array currently generates its value. The method in which this state is used is best described using an example.

Suppose the code segment in the upper left of Figure 4.18 is to be mapped to the combinational array in the bottom left of that figure. First, the ADD instruction enters the control generator. The two sources, r1 and r2, are accessed in the named variable table to determine which array nodes are currently generating their values. Both of these operands are not being produced, so they are mapped to the first two inputs. Next, the ADD operation is placed using the configuration update logic. This instruction needs to communicate with both of its inputs, and so it must be placed such that there is a communication path between it and the first two input nodes. Determining the available communication patterns within the subgraph is accomplished through a communication table. This table is indexed by node ID and returns a bit vector of the nodes that it can potentially communicate with. In order to determine which array cells an operation could be mapped to, the communication table values for the producers of the operation's sources are ANDed together . For example, the ADD operation has sources produced by Input 1 and Input 2. Input 1 communicates with node A and B, while Input 2 communicates with nodes A, B, C, and D. Thus, the ADD operation can be placed on nodes A or B. In this example, we greedily select node A, and update the configuration state to reflect that node A executes an ADD, and gets its inputs from nodes input 1 and input 2. With the configuration updated, the named variable state is updated to reflect that r1 and r2 are produced by nodes input1 and input 2, and r3 is now produced by node A.

The use of the communication table in configuration updates was not necessary for combinational arrays with full interconnect since any node could communicate with any other node in the previous row. Interestingly enough, use of the communication table actually simplifies control generation somewhat, despite the added layer of indirection in the mapping process. Since all the communication information is available statically, the logic in the control generator can be optimized to take advantage of the reduced possibilities in mapping. This benefit is similar to an instruction scheduler with two possibilities being faster than a scheduler with four possibilities, since the decision chain is shorter with only

two choices.

Returning to the example, once the ADD is mapped to the combinational array, the OR now needs to be mapped. First, the sources are looked up in the named variable table and r5 is mapped to Input 3. Next, the producers of the two sources are used to determine which nodes the OR can execute on; in this case those are C and D. The control generator greedily selects C and updates the configuration. Subsequently, the named state table is updated to show that r5 is produced by input 3, and r4 is produced by node C. Control generation then continues with the SUB instruction. At this point, a problem arises during configuration update because the producers for the SUB instruction (nodes A and C) have no successors in common. To remedy this, two MOV instructions for the sources are placed at the front of the instruction mapping queue with the SUB behind them. MOV r3, r3 is then mapped to node G, since r3 is produced by node A. Similarly the MOV r4, r4 is mapped to node H. When, the SUB comes to the head of the instruction mapping queue this time, the producers of r3 and r4 have node K in common, and the SUB is mapped onto that node. After all the subgraph operations are mapped onto the array, additional moves are inserted for the live out values. Live outs are determined by observing the instruction trace after the subgraph and keeping track of which operands generated in the subgraph are used before they are killed. Once move insertion is complete for live outs, the configuration data is written to the control cache, enabling execution of the accelerator.

The actual hardware implementation of the control generator is quite simple. The legality check is merely a comparison to ensure that the opcode of a given instruction is supported. The configuration update logic requires a lookup into the communication table, an AND of those table entries, and then a priority encoder to find a one (if any) bit that is set in the result. If no bits are set (i.e., no nodes are available), the configuration update generates one or more MOVE instructions based on which nodes generate the sources. If at any time the configuration logic cannot map an instruction, and adding MOVEs does not help, control generation is simply aborted.

It is important to note that because of the greedy mapping heuristic, it is possible to create patterns which can be executed on the combinational array, but cannot be mapped by the control generator. This is a fundamental problem with mapping to arrays with sparse

interconnect, since determining the mapping requires an intense search, which is much too complicated to do in hardware.

4.12.3 LUT-Based Subgraph Execution

Lookup-tables are another attractive structure for developing hardware accelerators. Because LUTs behave like truth tables, they have the capability to execute any number of bit-wise operations as one simple table lookup. The Programmable Carry Functional Unit (PCFU) [129] is an example of a hardware accelerator that takes advantage of this fact. The PCFU is capable of executing subgraphs with any number of bit-wise operations and a predefined number of additions/subtractions. While this may restrict the number of subgraphs that can be executed compared to the combinational array, the simple interconnect of the PCFU allows for lower latency, and lower hardware cost.

4.12.4 PCFU Control Generation

From a control generation standpoint, the key observation to take away from the PCFU design is that mapping subgraph onto this type of accelerator requires translating computation into several lookup-table entries. Using the control generator from Figure 4.16 as a template again, the named state for this control generator is a LUT value used to generate that register as a function of the input registers. The legality check for this accelerator ensures that opcodes in the subgraph are supported and that the maximum number of additions is not exceeded (recall carry generators are a fixed resource and a separate one is required for each addition). The configuration update logic takes the LUT entries for the source registers of an incoming instruction, and creates a new LUT entry describing how to compute the destination of that instruction as a function of the subgraph inputs. The p and g LUT values comprise the configuration state of this accelerator.

Figure 4.20 shows a high-level example of generating LUT entries for a dataflow subgraph. At the left of the figure is the subgraph to be mapped, and the bottom of the figure shows the named state at various stages of the LUT generation. The PCFU being targeted in this example can support two inputs and one addition operation, thus each output bit is

inst1: EOR r6,r1,r2
inst2: AND r7,r4,r5
inst3: ORR r12,r6,r7

r5	r4	r2	r1	(r6)
in3	in2	in1	in0	in0 ^ in1
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

r5	r4	r2	r1	(r7)
in3	in2	in1	in0	in3 & in2
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

r5	r4	r2	r1	(r12)
in3	in2	in1	in0	(in0 ^ in1) | (in3 & in2)
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

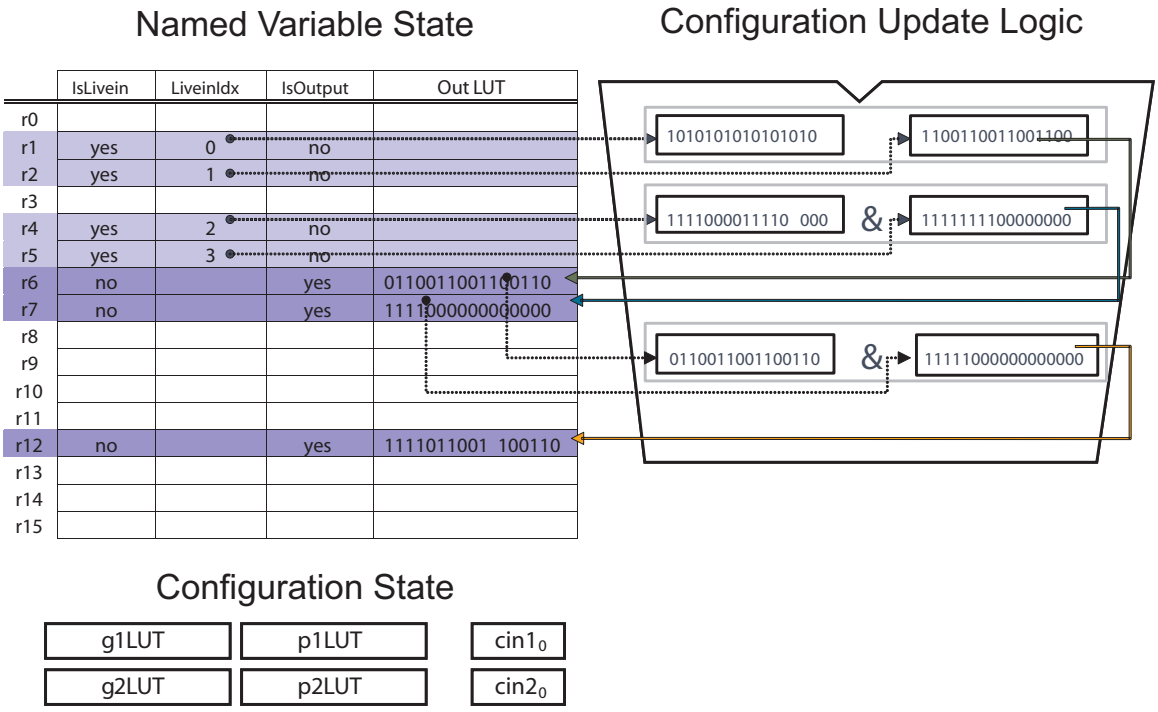


Figure 4.19: LUT generation example

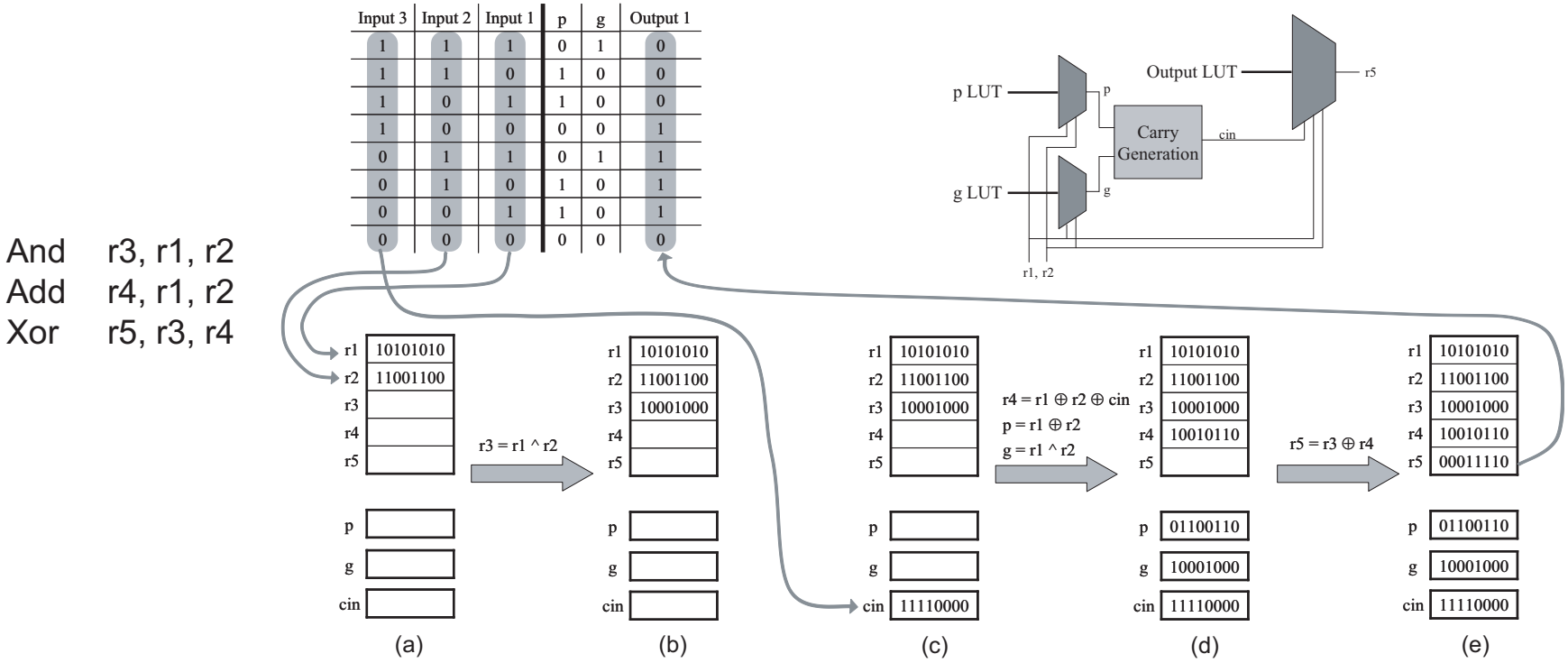


Figure 4.20: LUT entry generation example. Shown are the processing steps of the control generator that compute the LUT entries to implement the function defined by the assembly code sequence on the left.

a function of three input bits (the two inputs, and the carry signal from the addition). The three input bits imply that each LUT entry is eight bits in size, as $2^3 = 8$.

Subgraph mapping begins by looking at the And instruction in Figure 4.20. Initially, the two sources, r1 and r2, have no LUT value in the named state. Since we are interested in computing the output given all possible values for the inputs, named state r1 and r2 are assigned LUT entries which ensure that all possible combinations of one and zero interact (basically, the corresponding input columns of a regular truth table). This step essentially begins to construct a truth table. Since r1 and r2 are live-ins, the LUT entries assigned to them correspond to the input columns of a hypothetical truth table that outputs to named state r3.

Now that r1 and r2 have values in the named state, r3 is computed in configuration update as the And of those two values, and is shown in Figure 4.20 (b). To reiterate, the value of r3 in the named state table defines how to compute r3 given any values of r1 and r2, exactly like a truth table.

The subgraph mapping then moves onto the Add instruction. The output of this instruction is dependent on r1, r2, and also on the carry signal generated during addition. Recall that the carry-in signal is treated as an input to the computation so that we can leverage fast carry generation hardware, and so that each output bit is not dependent on the value of lower order input bits. Since the carry signal of this Add is an input, we assign it a value in the named state table that corresponds to a third input column of the hypothetical truth table. This step is shown in Figure 4.20 (c).

Now that the inputs are defined, the LUT entries for outputs of the Add operation must be computed. Recall that a bit-wise add operation is $r1_i \oplus r2_i \oplus cin_{i-1}$. Using the LUT entries for r1, r2, and cin, the configuration update logic computes this and places the result in named state r4. Note that because cin is defined as an input, the configuration update logic did not need to perform an addition, only two exclusive-ors, which makes the hardware very fast and simple.

Unlike the registers, the carry signal is generated using a carry propagation network. Thus, it is necessary to define p and g, the inputs to the carry network. Recall that $p = a \oplus b$ and $g = a \wedge b$. Using the values in the named state table for r1 and r2, the configuration

update logic simply computes the LUT values for p and g, and stores them into the named state table. The named state table after this step is shown in Figure 4.20 (d).

Mapping continues with the Xor instruction. As with the previous two instructions, we first check that all the inputs are defined. In this case, r3 and r4 already have valid values in the named state table and this instruction does not generate a carry signal. Next, the configuration update logic computes the LUT value of r5, by simply Xor-ing the LUT values of r3 and r4. This is shown in Figure 4.20 (e). Now that the output, r5, is defined as a function of r1, r2, and carry, we store the named state values of r5, p, and g as the configuration of this subgraph. This example shows how the control generator is able to perform logic mapping of a dataflow subgraph onto the PCFU substrate without the typical complexity associated with FPGA mapping.

Figure 4.19 shows a more detailed example of generating LUT entries for a subgraph of 3 instructions. The right portion of this figure shows the key components of a control generator for the PCFU. Note that since LUTs are simply the hardware equivalent of a truth table, LUT entries are $2^{numberofinputs}$ bits in size. In the example shown, the LUT entries are 16 bits wide, because the accelerator supports four inputs (corresponding to r1, r2, r4 and r5). The left portion of Figure 4.19 shows the truth tables that are conceptually constructed during control generation. These are shown only for clarity and are not part of the named or configuration state.

Each entry of the named state table has the following fields:

IsLivein This field is set to true when a register is live-in (r1, r2, r4 and r5 in this example).

LiveinIdx Each live-in register is given a unique live-in identifier. This identifier is used to determine the initial value of the LUT for this register.

IsOutput This field is marked true if the register is an output operand of a previously decoded instruction in the subgraph.

OutLUT Contains the LUT entry which computes the current value of a register as a function of the inputs.

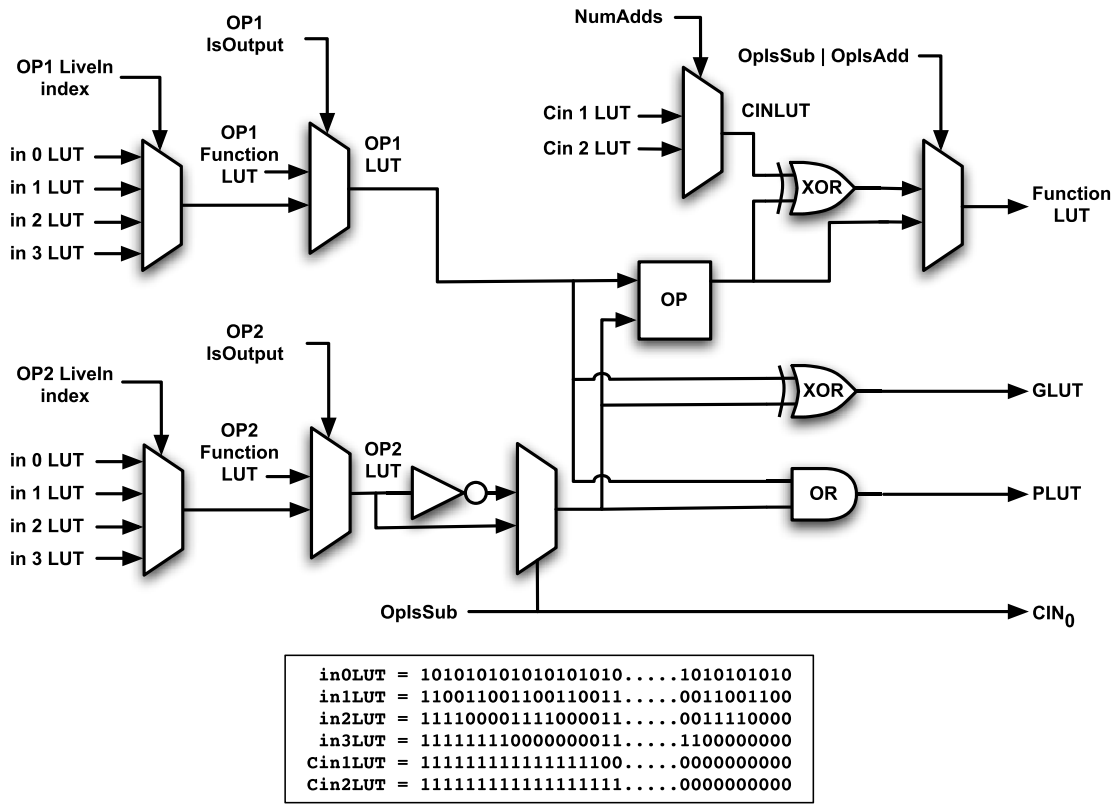


Figure 4.21: Structure of the PCFU configuration update logic

In the example in Figure 4.19, when the instruction `inst1` enters control generation, the corresponding `IsOutput` field is checked to see if a previous instruction produces an output in r1, because `IsOutput` is false, the `IsLivein` entry is set to true and `LiveinIdx` is given the index 0. Because this operand is live-in, the corresponding operand LUT entry is assigned a predefined value corresponding to the rightmost column of the truth table (column in0); similarly the second operand is assigned a LUT entry corresponding to the second column of the truth table (in1). Note that if a register operand is found to be already marked as live-in, the column given by `LiveinIdx` is assigned to the operand's LUT entry. The two operands' LUT entries are then fed to the configuration update logic. The resulting LUT entry, which corresponds to the XOR of the two input LUT entries, is stored in the output operand (r6) entry in the named state table, and the

corresponding `IsOutput` entry is marked as true. The LUT entry calculated corresponds to column r6, computed as the XOR of column r1 and r2. Similarly, instruction `inst2` is decoded and columns in3 and in4 of the truth tables are ANDed. The resulting LUT entry is stored in the (r7) entry of the named state table. Finally, when processing instruction `inst3`, the corresponding LUT entries are read from the named state table (because the `IsOutput` entry of the operands are marked as true) and sent to the configuration update logic to calculate the LUT entry for r12. Once all the instructions are processed, the live-out LUT entries and the p and g LUT entries are stored as the control for this subgraph.

As with the array of combinational logic, the hardware needed to generate control for the PCFU is quite simple. The legality check ensures that the operations to be mapped are bit-wise, or that the maximum number of addition/subtraction operations in a subgraph are not exceeded. The configuration update logic is for this control generator is shown in Figure 4.21. Note that the each operation performed on the LUT entries is bit-wise, making the critical path of this unit only 8 gates long. The named variable state table is the largest portion of this control generator. Since each register is expressed as a function of the inputs, it need 2^{inputs} bits per register for LUT entries. Beyond this, additional bits per register are needed to determine whether or not each register is an input or an output to the subgraph. The input bits are needed to ensure that subgraph inputs are routed to their appropriate LUT selection slot, and the output bits are needed to select which LUT entries are used in the PCFU configuration.

4.13 Evaluation of Control Generators

To perform this evaluation, we use a version of the Trimaran compiler [121] ported to the ARM instruction set. During compilation, selected subgraphs were outlined as function calls and assembled/linked using the GNU tool chain. The resultant binaries were then run on a version of SimpleScalar [9] configured to model an ARM 926EJ-S processor [5].

Table 4.2 shows the results from synthesizing various subgraph accelerator designs, and there associated control generators. These designs were synthesized (including place-and-route) using Synopsys tools with an Artisan standard cell library in 0.13μ. The latencies

	Latency (ns)	**Area (mm^2)**	**Area (% of ARM926EJ-S)**
Comb. Array - Full Interconnect [28] (Figure 4.17 (a))	5.52	0.225	10.26
Comb. Array - Medium Interconnect (Figure 4.17 (b))	5.34	0.189	8.62
Comb. Array - Sparse Interconnect (Figure 4.17 (c))	5.18	0.173	7.88
PCFU (4 inputs -2 outputs)	5.02	0.089	4.08
PCFU (4 inputs)	4.78	0.068	3.11
PCFU (3 inputs)	4.32	0.057	2.59
Array Control Gen. - Full Interconnect	5.03	0.045	2.08
Array Control Gen. - Sparse	4.50	0.038	1.75
LUT-based Control Gen.	4.07	0.162	7.36

Table 4.2: Synthesis Results for Dynamic Control Generators

shown are critical path of the design. Areas are given both in mm^2 and as the percentage area of an ARM926EJ-S core.

There are several interesting trends in the synthesis results to notice. First, pruning the interconnect did not reduce the latency of the combinational arrays as much as we had expected. Moving from full interconnect to sparse interconnect only resulted in 6.6% reduction in latency. However, pruning the interconnect did result in a 30% reduction in die area. As discussed earlier, pruning the interconnect also resulted in reduced latency and area of the control generator.

When comparing with the combinational arrays with LUT-based execution units, the first thing that jumps out is the relative size difference. This was to be expected, as the combinational arrays replicate function units many times, while the LUT-based unit only has a few lookup tables and two relatively simple carry-propagation networks. The price for simplicity in the execution unit is payed for with the relative complexity in control generation in terms of hardware cost, though. We found the LUT-based control generator to be almost four times the size of the combinational array control generator. The primary reason for this is the significant amount of additional state necessary to generate to LUT entries in the named state table. Despite the larger area, the LUT-based control generator had a shorter critical path than the combinational array one. This is because the LUT-based unit does not have the long decision networks associated with scheduling operations to

various nodes in the combinational array.

Overall, the synthesis results show that control generators consume very little area overhead, and are fast enough to fit into existing embedded processors without affecting the clock cycle.

4.14 Summary

The drive to improve the performance and efficiency of processors will inevitably lead to the use of more and more computation accelerators. However the traditional methods for utilizing these accelerators, by changing the instruction set, are very costly. In this chapter we have presented a generalized framework to dynamically generate control for these accelerators using dataflow subgraphs expressed in the instruction set of the baseline processor. Using these systems enables processors without accelerators to run the applications, and processors with accelerators to run more efficiently.

Through the use of hardware synthesis, this chapter demonstrated the feasibility of the proposed control generation framework on a variety of different computation accelerators, including arrays of combinational logic and lookup-table based accelerators. The overhead of dynamic mapping systems is typically less than 10%, even on simple embedded cores. Transparent instruction set customization is an effective way to improve computation efficiency without the overheads typically associated with instruction set modification.

CHAPTER 5

Compilation Techniques for Acyclic Accelerators

5.1 Introduction

Previous chapters have discussed the design and utilization of several types of acyclic accelerators. The utilization techniques rely on statically identifying subgraphs to execute on an accelerator, and then dynamically mapping those subgraphs to the micro-architecture. An often overlooked challenge in this area, and the focus of this chapter, is the compiler support that underlies static identification. The compiler has two major tasks when compiling toward accelerators. First, it must identify candidate subgraphs in the target application that are functionally executable on the accelerator. This is essentially a subgraph isomorphism problem. The second task is to select which candidate subgraphs to actually execute on the computation accelerator. Candidates often overlap, thus the compiler must select a subset to maximize performance gain. This task is essentially a graph covering problem.

Most prior solutions employ a greedy compiler approach for both subgraph identification and selection [17, 60]. With this approach, a seed operation is selected and a subgraph compatible with the accelerator is grown by iteratively including connected operations. As with all greedy approaches, this approach can achieve sub-optimal solutions in both identification and selection. Further, disjoint subgraphs cannot be identified. However, for small accelerators, such as 3-1 ALUs, this approach is sufficient due to the simple nature of compatible subgraphs. The greedy approach breaks down for larger accelerators where correspondingly larger subgraphs must be identified. As a result, others have proposed us-

ing exact methods for subgraph isomorphism and covering [76, 80, 101]. These methods grow exponentially in subgraph size, region size (the unit of operations analyzed by the compiler), or both. As a result, exact methods can suffer from excessive compilation times for moderate to large applications and hence may not be practically deployable.

In this chapter, we propose an approach for compiler subgraph mapping that combines exact methods with a set of intelligent pruning techniques. Pruning ensures the proposed algorithms are scalable in both application and accelerator size to provide practical compilation times. The approach has three distinct phases. First, potential subgraphs are identified using bounded enumeration. Subgraph isomorphism is then used to remove candidates that are not compatible with the computation acceleration. Finally, unate covering is used to select subgraphs that will be executed on the accelerator.

This chapter makes the following three contributions:

- We collect and describe state of the art algorithms for accelerator compilation.
- New algorithms for identifying and mapping subgraphs optimally with intelligent pruning mechanisms are proposed.
- The new algorithms are evaluated for both performance and compilation time across a variety of accelerator designs, and the results are compared to a traditional greedy approach.

5.2 Problem Statement and Related Work

Compiling an application to make use of computation accelerators boils down to two steps: *enumerating* portions of the application's dataflow graph (DFG) that can be executed on the accelerator, and *selecting* which portions to accelerate.

Enumeration consists of generating a set of subgraphs from a given DFG, and determining if they can run on an accelerator. Generating a set of subgraphs is difficult, because the number of subgraphs grows exponentially with the size of the DFG. Determining if the subgraphs can run on an accelerator, i.e., determining if they perform the same computation, is essentially equivalence checking, which is NP-complete. The problem is further compli-

cated if the accelerators perform a superset of the desired computation (e.g., an accelerator for dot-products could also accelerate multiply-accumulates in an application).

Selecting which subgraphs to accelerate is also difficult. Typically, the selection problem is formulated to push as much computation as possible onto the accelerators, while minimizing overlap between subgraphs. That is, given a set of enumerated subgraphs, find the group which covers the largest portion of the DFG while minimizing the number of nodes appearing in multiple subgraphs. This problem is also NP-complete and is quite similar to the well known technology mapping problem in VLSI design. Clearly, mapping applications to subgraphs is a challenging compilation problem.

To side step the problem, the vast majority of previous work relies on hand coding or greedy heuristics. Work on automated accelerator design typically does not discuss strategies for utilizing the accelerators with compilers. Work by Hu [60] is typical of the greedy solutions: a seed node is selected in the DFG, and is grows along dataflow edges. The compiler then replaces that subgraph and repeats the process. Here enumeration consists of finding a seed and growing it, while selection is implicit (any subgraph that is enumerated is automatically selected). Other previous work [112] performs more thorough enumeration, but still utilizes greedy selection.

More thorough, traditional code generation methods for tackling subgraph mapping use a tree covering approach [3]. In this approach, all computation subgraphs potentially supported by the accelerator must be constructed a priori. During compilation, the DFG is split into several trees. The trees are then covered by the computation subgraphs using an algorithm that minimizes the number of computation subgraphs used. The purpose behind splitting the DFG into trees first is that there are linear time algorithms to optimally cover trees, making the process very quick.

The major problem with this method is that many DFGs and accelerators are not trees. It is shown in [76] that tree covering methods can yield suboptimal results, particularly in the presence of irregular computation commonly targeted by embedded systems. To overcome this, [76] proposes splitting all instructions into “register-transfer” primitives and recombining the primitives in an optimal manner using integer programming. Work by Liao [80] attacked the same problem, and developed an optimal solution for DFG covering

by augmenting a binate covering formulation. While both of these solutions are optimal, they also have worst case exponential runtime, and do not report how long their algorithms take.

Another major problem with previously mentioned approaches is that they also require permissible accelerator subgraphs to be enumerated a priori. If an accelerator supports a wide range of computations, such as an ALU pipeline, this can cause an explosion in runtime.

Research in [101] describes a different way to look at the accelerator mapping problem. In this work, an application is initially decomposed into an algebraic polynomial expression that is functionally equivalent to the original application. Next, the polynomial is manipulated symbolically in an attempt to use accelerators as best as possible. For example, a polynomial could be expanded using function identities (e.g., adding 0 to a value) to better fit an accelerator. This enables the algorithm in [101] to utilize subgraphs where the accelerator performs a superset of the desired computation. As with previous solutions, though, this technique also has exponential worst-case runtime. Additionally, handling bit-wise operations, e.g. XOR, using polynomials is difficult.

In this work, we present compilation techniques to exploit acyclic computation accelerators. These techniques produce higher quality code than greedy heuristics, do not require a priori enumeration of permissible accelerator subgraphs, and are scalable to large applications.

5.3 Compilation for Acyclic Accelerators

In this section, we present two different approaches for compiling to acyclic accelerators. The first approach, *greedy enumeration - immediate selection*, is the most commonly used approach today. This method generates a set of subgraphs by greedily adding vertices to a seed vertex from the dataflow graph. Once the subgraphs are grown, they are immediately replaced in the application, thus the name immediate selection. The second approach, *full enumeration - unate covering selection*, is our contribution. This approach generates all possible dataflow subgraphs subject to certain constraints of the targeted accelerator.

The set of subgraphs is then pruned down using subgraph isomorphism, and finally unate covering selects which subgraphs end up being executed on the accelerator.

5.3.1 Greedy Enumeration - Immediate Selection

Greedy enumeration - immediate selection, or greedy algorithms for short, is the standard method used to target acyclic accelerators, e.g. in [17, 59]. The greedy algorithm consists of two phases: seed selection and subgraph growth. Using a basic block as input, the greedy algorithm selects an operation as a seed and tries to expand that seed by iterating over dataflow edges. After growing one seed as much as possible, the subgraph is replaced. Next, another seed is selected, and the same steps will be repeated. The algorithm finishes when no more seeds are available for growing.

The first step in the greedy algorithm, seed selection, can be performed in several different ways. For example, operations closer to the critical path can be chosen as seeds before less critical operations. Alternately, long latency operations can be selected before shorter operations. In our experiments, *changing seed selection order made very little difference in the results of the greedy algorithm*. In the results presented in this chapter, seeds were chosen in topological order from the dataflow graph.

After choosing a seed, a subgraph consisting only of that operation is formed. The algorithm then enters its second phase, subgraph growth, trying to expand this subgraph. Neighbors of the seed operation will be temporarily added to the subgraph one at the time. If this temporary subgraph is executable on the accelerator, then the new node permanently becomes part of the subgraph. If the temporary subgraph is not executable, then the newly added node will be removed. When it is no longer possible to add neighbors to the subgraph, it is immediately replaced in the application, and a new seed is selected from operations not already appearing in a subgraph.

An example of the greedy algorithm is shown in Figure 5.1. Figure 5.1 B is a DFG from the g721encode benchmark, used in examples throughout the chapter. Figure 5.1 A shows the acyclic accelerator targeted in all of the examples. This accelerator, similar to one proposed in [30], has 4 inputs, 2 outputs, and 15 function units organized in 4 rows.

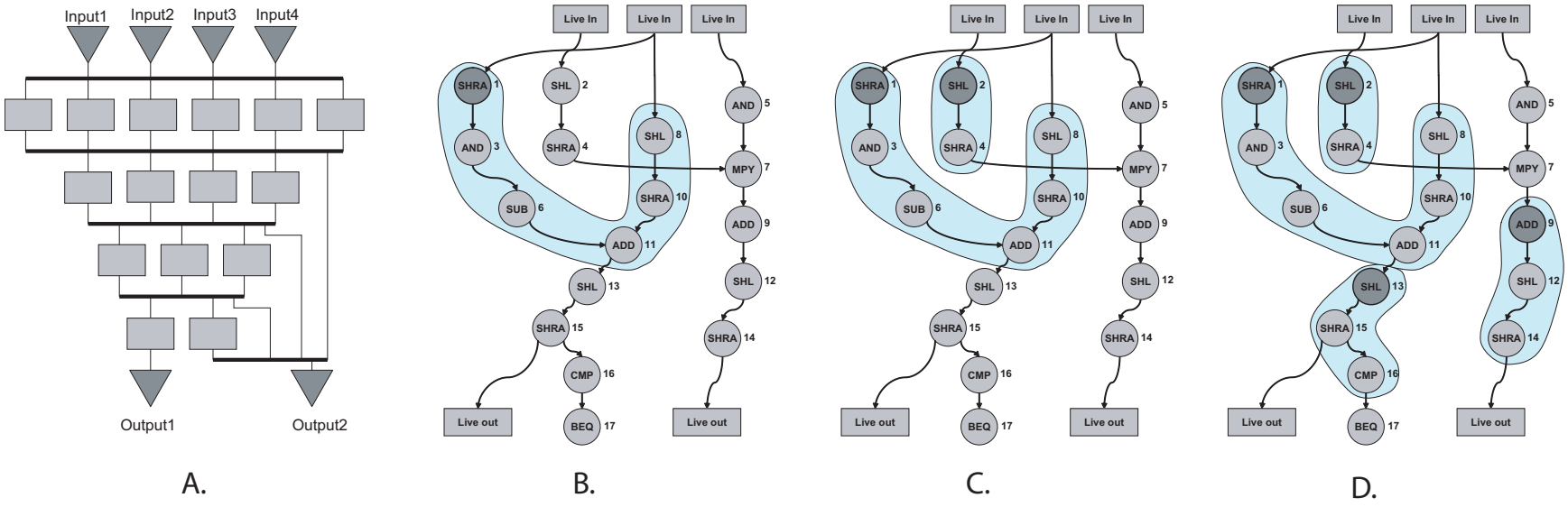


Figure 5.1: A. An acyclic accelerator from [30] targeted in examples. B. The first step in a greedy mapping algorithm on a basic block from g721encode. C. The second step and D. final step in the greedy mapping algorithm.

The function units in each row can communicate with function units in subsequent rows, meaning computations with dependence heights of up to 4 are supported. These function units support the complete set of addition, subtraction, and bit-wise operators on two inputs.

Figure 5.1 B highlights the first subgraph enumerated using the greedy method. Operation 1 is chosen as the first seed. The subgraph then greedily adds neighbor operations 3, 6, and 11. After adding operation 11, 13 can not be added since that would create a subgraph with dependence height 5, which is not supported by the accelerator. Operations 11's neighbors 10 and 8 can be added, though, resulting in the final subgraph shown in Figure 5.1 B. The process then repeats with operation 2 as a seed node. This subgraph is grown along dataflow edges until it reaches operation 7, a multiply, which is not supported by the accelerator. The final greedy mapping of the DFG is shown in Figure 5.1 D. Assuming each operation and the accelerator each take one cycle to execute, this mapping would yield a speedup of $\frac{17}{3+4} = 2.43$, since there are 3 unaccelerated operations and 4 accelerated subgraphs.

5.3.2 Full Enumeration - Unate Covering Selection

Greedy subgraph mappers have proven reasonably effective in many previous works. Certain combinations of greedy algorithms with more thorough strategies have also proven effective [112]. In this section, we describe the full enumeration - unate covering selection, or FEU, algorithm, which solves the mapping problem using exact formulations. This effectively avoids local minima that inherently cause greedy algorithms to fail. When the exact formulations are intractable, the FEU algorithm intelligently reduces the search space to workable levels.

There are three main phases to the FEU algorithm: Enumeration, Pruning, and Covering. Enumeration generates a set of all subgraphs within a DFG subject to input/output constraints of the targeted accelerator. Pruning takes the set of subgraphs and performs additional checks based on functionality and interconnect to determine if the subgraphs actually can be executed on the targeted accelerator. Once unusable subgraphs are pruned, unate covering is used to select the best set of subgraph instances for the application being

compiled.

Individually, each of these steps either grows exponentially with the size of the input (enumeration) or is NP-Complete [46, 48] (pruning and covering). This has lead most researchers to opt for (typically) linear-time greedy solutions. In the remainder of this section, we will demonstrate that with careful design, each of these problems can be made tractable for most practical cases in accelerator compilation. Additionally, we demonstrate in Section 5.4 that using more powerful algorithms yields noticeable performance improvements in code generated over the standard greedy approaches.

5.3.2.1 Full Enumeration

The first step of our compilation algorithm, enumeration, generates a set of dataflow subgraphs that can potentially be run on a targeted accelerator. The primary reason for enumerating subgraphs and then later pruning them is that it is much faster than performing both steps at once. Very fast techniques for finding high-quality subgraphs for acceleration have been widely developed in the past few years, e.g. [7, 15, 27], and this strategy allows us to take advantage of them.

Tractable subgraph enumeration is clearly a difficult problem. In the most general sense, each operation in a DFG could either be included or excluded in a potential subgraph instance, yielding 2^N potential candidates. Because of space restrictions, the large body of previous work, and the relative complexity of pruning techniques, we will only describe how to efficiently enumerate subgraphs at a high level.

Dataflow subgraph enumeration, as described in [7], can be thought of as a binary tree, where each level of the tree represents an operation (op for short), and each branch in the tree represents whether or not to include that op in a subgraph. The leaves of the tree represent all possible subgraphs for a DFG. There are many keys to make full exploration of this tree tractable.

The most important pruning technique is based on input/output restrictions of the accelerator. Using the DFG from Figure 5.1 B as an example, if a targeted accelerator only supported 2 inputs, then any candidate subgraph including ops 1, 2, and 5 would be infeasible. Enumeration can be bounded for each branch of the tree that includes all of those

ops. Likewise, pruning for outputs greatly reduces the search space. Care must be taken to avoid prematurely pruning the search space, though. For example, a subgraph with ops 6 and 10 would appear to have 2 outputs; however, if op 11 is included, then subgraph 6, 10, 11 only has 1 output, perhaps making it feasible.

Another important pruning technique is excluding candidates with values that leave and then reenter the subgraph. Using Figure 5.1 B as an example again, this filter would prune the search space of any subgraph that included ops 1 and 6 but excluded op 3. Subgraph 1, 6 could not be run on an accelerator since the output of 1 is used to calculate an input to 6.

These techniques make subgraph enumeration practical for the vast majority of blocks within applications; there are some instances where additional steps are needed. In these cases, the DFG is heuristically partitioned into several sub-blocks, which are then enumerated. The implication of partitioning is that no candidate subgraphs can cross the boundary (i.e., it cannot have ops in multiple partitions). Edges are heuristically weighted to guide the partitioner so that is does not unnecessarily cut edges for important subgraphs. For example, if the targeted accelerator did not support multiplication, then all the edges to and from op 7 in Figure 5.1 B would be given weight 0, since the ops on either side of the edges could never be in a feasible candidate. Edges bordering memory operations are also given weight 0 whenever the accelerator does not support memory access. All other edges are given weights based on characteristics such as whether or not they are on the critical path. Previous work [27] demonstrated that this heuristic partitioning is an effective way to prune the enumeration space without unnecessarily removing useful subgraph candidates.

5.3.2.2 Pruning through Subgraph Isomorphism

Pruning is the next step after enumeration generates potential subgraphs to execute on the accelerator. The purpose of pruning is to ensure that candidates can actually be executed on the accelerator. This takes into account functionality and connectivity issues that were ignored during enumeration. Pruning occurs after enumeration because these checks are either not possible to perform on partial candidates or too heavy weight to test in the middle of filling in the enumeration search tree.

The method employed to determine that subgraphs can execute on an accelerator is

based on subgraph isomorphism. Loosely stated, subgraph isomorphism determines whether or not a subset of the nodes in a particular graph are equivalent to a separate graph. In this case, a graph representing the hardware structure is constructed, and we attempt to find a subset of hardware vertices that can create a computation equivalent to the subgraph created in enumeration. If we find such a subset, then the dataflow subgraph is capable of being executed on the accelerator.

There are several pros and cons to pruning based on subgraph isomorphism. One benefit is that, as with enumeration, a great deal of related work (e.g. [70, 122]) has looked at developing heuristics to efficiently solve subgraph isomorphism the problem. We leverage and improve upon these prior techniques in this work. An additional benefit is that previous work [124] has shown it is possible to automatically generate hardware subgraphs from a microarchitectural specification. This means that a compiler targeting accelerators could potentially be retargeted by simply feeding it a hardware description of the targeted accelerator(s).

The main weakness of isomorphism-based pruning is that it is not a true equivalence check. That is, the algorithm only checks that nodes used to represent computation form equivalent graphs, not that they are equivalent computations. For instance, if a DFG represented a multiplication by 10 as a left-shift by 3 bits, a left-shift by 1 bit, and an addition of those two results, then this would not match an accelerator with a multiplier. In order to recognize multiple graphs that perform the same computation, pruning would have to perform a full equivalence check, typically using BDDs [22] or their relatives (ADDs, BMDs, etc.). This is far more computationally demanding than isomorphism for accelerators of practical size, although an interesting avenue for future work.

The implications of this drawback are twofold. First, the compiler is at the mercy of the software writer to a certain extent. If the algorithm is described in software differently than it is represented in the hardware graph, then the compiler will be unable to accelerate it. Second, accelerator hardware structures that do not map directly to a single node in the DFG are difficult to utilize. For example, a lookup-table is capable of executing any number of consecutive bit-wise operations from a dataflow graph. Because of this, there is no equivalent (finite) hardware graph that can represent this computational structure.

This drawback affects both full-enumeration-based and greedy-based compilation algorithms, and leaves room for improvement. However equivalence-based algorithms have proven intractable to this point.

Subgraph Isomorphism Algorithm: The algorithm used to determine isomorphism, Algorithm 5.1, is based on the backtracking search strategy described in [70], which was itself adapted from [122]. The basic idea is to recursively assign one vertex from S', the dataflow subgraph, to a corresponding vertex in T, the hardware graph, and check to ensure that the corresponding edges exist in both graphs whenever a new node is assigned. In order for this algorithm to be computationally feasible, a number of steps are taken to prune the search space.

Algorithm 5.1 takes the two graphs $S' = (V', E')$ and $T = (W, F)$ as input. In this formulation, V' represents operations in the subgraph, E' dataflow edges in the subgraph, W FUs in the accelerator, and F wires connecting those FUs. Initially, a group of sets, M, are calculated such that M_i contains all vertices in W that are of the same computation type as v_i. Essentially this step creates a set of candidate nodes in T that each node in S' can be mapped to. For example, if v_1 was an ADD node, M_1 would contain all hardware nodes with addition capabilities in W. This process corresponds to lines 2 - 6 of Algorithm 5.1. This information is passed to the procedure $AssignVertex$, along with the mapping function, $x()$, and the vertex number to be mapped.

The $AssignVertex()$ procedure iterates over the set of possible nodes (line 13 in Algorithm 5.1) testing that every edge in E' has a corresponding edge in F for the nodes that have already been mapped (lines 15 - 17). Assuming that the edges match, $x()$ is updated and the sets of potential matches, M, is updated to reflect the new information. This pruning of the search space is critical to avoiding an exponential explosion of runtime.

Two techniques are used to remove nodes from M after a node assignment. The first, lines 25 - 26, looks at all vertices in V' not yet assigned and checks to see if there is an edge in E' connected to the node just assigned, v'_{vertex}. If such an edge exists, any nodes in M that do not have a corresponding edge in F connected with $x(v'_{vertex})$ can be removed from the search space. The second pruning technique (lines 28 - 29) leverages the fact that that we are dealing with directed acyclic graphs. When creating S' and T, we impose the

```
1  Input: S' = (V', E'), T = (W, F)
2  foreach v'_i ∈ V' do
3      foreach w_j ∈ W do
4          if v'_i is equivalent to w_j then
5              if dependence_height(v'_i) ≤ dependence_height(w_j) then
6                  M_i = M_i + w_j
               end
           end
       end
   end
7  Call AssignVertex(M, x, 1)

   Procedure AssignVertex(M, x, vertex)
8  if vertex > |S'| then
9      if Subgraph outputs map then
10         return ISOMORPHIC
       end
11     else
12         return NOT ISOMORPHIC
       end
   end
13 foreach m_i ∈ M_vertex do
14     edges_match = true
15     for j = 1..vertex do
16         if e_{v'_j, v'_vertex} ∈ E' and e_{x(v'_j), m_i} ∉ F then
17             edges_match = false
           end
       end
18     if edges_match then
19         set x(v'_vertex) = m_i
20         M' = M
21         assignment_works = true
22         for j = vertex + 1..|V'| do
23             M'_j = M'_j − m_i
24             foreach m_k ∈ M'_j do
25                 if e_{v'_vertex, v'_j} ∈ E' and e_{x(v'_vertex), m_k} ∉ F then
26                     M'_j = M'_j − m_k
                   end
27                 else
28                     if e_{v'_vertex, v'_j} ∈ E' and k < i then
29                         M'_j = M'_j − m_k
                       end
                   end
               end
30             if |M'_j| == 0 then
31                 assignment_works = false
               end
           end
32         if assignment_works then
33             result = call AssignVertex(M', x, vertex + 1)
34             if result == ISOMORPHIC then
35                 return ISOMORPHIC
               end
           end
       end
   end
36 return NOT ISOMORPHIC
```

Algorithm 5.1: Subgraph isomorphism algorithm

restriction that the vertices must be topologically sorted within the sets V' and W. That is to say, $\forall$ vertices i, j such that $i > j, e_{i,j} \notin E$. In other words, there are no edges from vertices with higher order numbers to vertices with lower order numbers. This restriction allows us to remove any vertex from M which has a lower order number than the currently assigned order number, since no such backward edge can exist in F. If at any point during pruning, the size of the candidate set falls to zero (line 30), then it is no longer necessary to examine this part of the search tree. These simple pruning techniques turn an intractable problem into one that is solved much faster than instruction scheduling in our compiler infrastructure.

After pruning the search space, $AssignVertex$ is recursively called to assign the next vertex using the reduced search space, M'. This is continued until all nodes in S' map to corresponding nodes in T though the function $x()$, or it is proven that no such mapping exists. Once a mapping is found, it is still necessary to ensure that the subgraph outputs map onto the targeted accelerator (line 9). This is done using the Dijkstra's algorithm to find the shortest path between nodes producing the outputs and output ports. If this final check passes, then the subgraph can indeed execute on the targeted accelerator.

Improvements Over Previous Work: There are three main algorithmic improvements over previous proposed subgraph isomorphism algorithms. First, as previously mentioned, vertex numbers are assigned topologically to ensure that if an edge exists, then the source number is less than the destination number. This dramatically reduces the sets of potential candidates, M, shown in lines 28 and 29. A second improvement prunes the candidate sets by using dependence height of the candidates (line 5 of Algorithm 5.1). Dependence height refers to the maximum sized chain of operations that must precede a particular operation in a graph. For example, in Figure 5.1 B, node 10 has a dependence height of 1 since 8 must precede it, and node 11 has a dependence height of 3, since the chain 1-3-6 must precede it. When creating a set of candidates for node 11 in the representative hardware graph, we know that skipping any nodes with dependence height less than 3 will not affect the solution. This optimization also relies on the acyclic nature of the graphs we are matching, and has a dramatic impact on the overall algorithm runtime. The last optimization developed relates to the order in which nodes are assigned. Note that in $AssignVertex$, pruning of

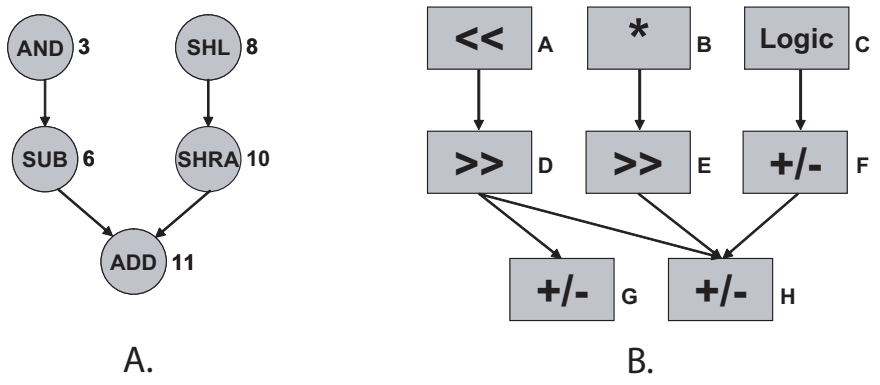


Figure 5.2: A. Subgraph from Figure 5.1 A to be tested for subgraph isomorphism, B. hardware accelerator being targeted

M occurs when edges do not match up in the current assign, $x()$. Thus, it is in our interest to make these comparisons as high in the search tree as possible. This is accomplished by assigning vertices in order determined by a depth first search (not shown in Algorithm 5.1). Unlike the previous two optimizations, this technique is applicable for any style graph, not just directed-acyclic graphs. These three optimizations contribute to make subgraph isomorphism a tractable way to determine whether a dataflow subgraph can execute on a hardware accelerator.

Subgraph Isomorphism Example: Algorithm 5.1 is complicated and we will hopefully clarify it through the example in Figure 5.2. Here, the dataflow subgraph in Figure 5.2 A (from Figure 5.1 B) is checked for subgraph isomorphism on the accelerator graph in Figure 5.2 B. First, a set of candidates in Figure 5.2 B is constructed for each vertex in Figure 5.2 A. This corresponds to M in the algorithm. Examining vertex 3, we see that only hardware vertex C can execute logic operations, so $M_3 = \{C\}$. Likewise, $M_6 = \{F, G, H\}$, since any of those hardware vertices could execute the subtraction. The candidate set of vertex 11, $M_{11} = \{G, H\}$ demonstrates the dependence height pruning; F can not be in the solution space because there is only one hardware vertex preceding it. The remaining two sets, $M_8 = \{A\}$ and $M_{10} = \{D, E\}$ are as would be expected.

After the candidate sets are computed, a depth first search is performed (irrelevant of edge directions) to determine the order in which to assign vertices. In this example, the

assignment order will be 3, 6, 11, 10, and 8, although this ordering is irrelevant for correctness. $AssignVertex()$ is then called for node 3. The algorithm iterates over the set of candidates, M_3, and updates M for neighbor vertices. In this case, since vertex 6 neighbors vertex 3, M_6 can remove candidates G and H from its set, since neither of those vertices are neighbors of C. Next, $AssignVertex()$ is recursively called to map vertex 6. The algorithm maps vertex 6 to F, since that is the only possibility in M_6. lines 15-17 check to make sure that since there is an edge from vertices 3 to 6, that there is also an edge from C to F. Vertex G is removed from M_{11}, since there is no edge from F to G, and again $AssignVertex()$ is called for vertex 11. Vertex 11 is mapped to H, and 10 is mapped to E similarly to the previous two nodes. However, once 10 is mapped to E, then the candidate set M_8 becomes empty, since there is no edge from A to E. This bounds the recursion of $AssignVertex()$ which then tries another assignment for vertex 10, D. Using this mapping, vertex 8 can be assigned to A, which will complete the mapping, and prove that there is a subgraph of Figure 5.2 B that is isomorphic to Figure 5.2 A.

5.3.2.3 Selection using Unate Covering

Now that we have a set of subgraphs that *can* execute on the accelerator, it is necessary to select which ones *to* execute on the accelerator. In standard greedy solutions this step is implicit within enumeration: each enumerated subgraph is automatically selected. However, greedy selection can also be performed in conjunction with full enumeration algorithms, e.g., in [112]. Greedy selection algorithms, typically map the largest subgraph onto the application, remove all overlapping subgraphs from the consideration, and then repeat this process until no more candidates remain. The problem with this technique is that it will provide suboptimal results whenever the largest subgraph is not part of the best solution.

Instead of a greedy heuristic, we propose solving the selection problem by converting it to a unate covering. Informally speaking, unate covering problems operate on a Boolean matrix, M, where the rows represent vertices in a DFG, and the columns represent subgraphs; if the value of $M_{i,j}$ is true, this means that operation i occurs in subgraph j. Traditionally, the goal of unate covering is to find a set of columns (or subgraphs) with

```
1  Input: boolean matrix M, where M_{i,j} = true if op i is in subgraph j
2  Output: A vector x, x_i ∈ {0,1}^n, where Mx = (1,1,1,...,1)^T and Σ_{i=1}^n x_i is minimized

3  Sort columns of M in order of decreasing size
4  Call Cover(1, true, M, x)
5  Call Cover(1, false, M, x)

   Procedure Cover(subgraph, add_subgraph, M, x)
6  if add_subgraph then
7      if (Mx&&(M_{1,subgraph}, M_{2,subgraph}, ...M_{m,subgraph})^T) ≠ (0,0,0...0)^T then
           // Subgraph overlaps with the partial solution.
8          return
       end
9      x_subgraph = 1
10     if Mx == (1,1,1,...1)^T then
11         if Σ_{i=1}^n x_i < fewest_subgraphs then
12             fewest_subgraphs = Σ_{i=1}^n x_i
13             best_solution = x
           end
           // Found a complete cover.
14         return
       end
   end
15 if subgraph + 1 > n then
       // Did not find a complete cover after examining all subgraphs.
16     return
   end
17 if Σ_{i=1}^n x_i + (m − Σ_{i=1}^n (Mx)_i) / (Σ_{i=1}^m M_{i,subgraph}) ≥ fewest_subgraphs then
       // The current solution cannot possibly be the best.
18     return
   end
19 Call Cover(subgraph + 1, true, M, x)
20 Call Cover(subgraph + 1, false, M, x)
```

Algorithm 5.2: Unate covering selection algorithm

minimal cost, such that each operation is covered at least once. In this formulation, the cost of a subgraph could be a variety of things, such as the number of cycles needed to execute on a particular accelerator, or the power consumed by a subgraph. As with using subgraph-isomorphism for the pruning algorithm, unate covering was chosen for selection because there is much prior work [36, 48] that can be leveraged to make this problem tractable.

Before discussing the details of our unate covering algorithm, Algorithm 5.2, it is important to point out one difference between this and standard unate covering formulations. Traditionally, unate covering allows an operation to appear in multiple subgraphs in the final code. However, we have made the decision to disallow this possibility. Allowing an operation to appear in multiple subgraphs essentially replicates the computation and will unnecessarily increase power consumption. The downside is that disallowing overlapping subgraphs can hurt application performance in multi-issue processors, and actually makes the covering search space much larger. Performance loss can occur because the first operation in a subgraph has to wait for all subgraph inputs to be ready before being executed. The covering search space becomes larger, because many techniques to prune the space, such as row and column dominance, no longer work if overlap is not allowed. Despite the changes resulting in a large search space, the runtimes of our unate covering formulation are quite reasonable for practical inputs, and the resulting code will be more suitable for embedded systems.

Unate Covering Algorithm: The algorithm used to perform unate covering based selection is shown in Algorithm 5.2. As previously mentioned, input to the algorithm is a m by n Boolean matrix, where rows correspond to operations and columns to subgraphs. The output of this algorithm (line 2) is a vector, x, where $x_5 = 1$ means that subgraph 5 is in the optimal cover. The constraint $Mx = (1, 1, 1, ..., 1)^T$ ensures that each operation is covered by exactly one subgraph. Note that the standard unate covering constraint, which allows overlap, is $Mx \geq (1, 1, 1, ..., 1)^T$. To ensure that a solution is feasible, each individual node is inserted into M as a subgraph which covers only one operation. Once M is constructed, the columns are sorted in decreasing order, and a standard branch-and-bound algorithm, $Cover()$, is called.

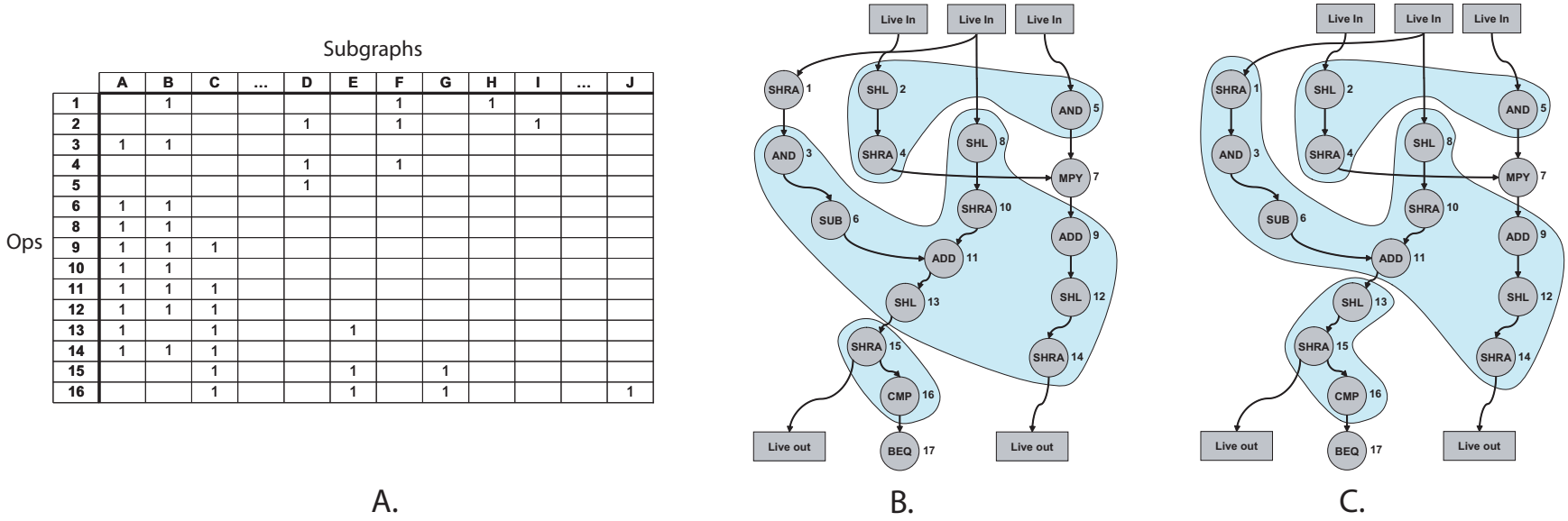
Inside the function $Cover()$, one subgraph is considered for addition to the current cover, x. Line 7 in Algorithm 5.2 tests to see if there is any overlap between the current cover and the candidate subgraph. The Mx matrix multiplication creates a column vector of the current set of ops that are covered, and $M_{i,subgraph}$ is the set of ops covered by $subgraph$. Assuming there is no overlap, line 9 adds $subgraph$ to the current cover, and then the cover is tested to see if all ops are covered (line 10). If a complete solution exists, the total number

of subgraphs is calculated, and if it is the fewest yet seen, then this cover is recorder as being the best. Note that if there were multiple accelerators in the targeted processor, the notion of what constitutes the 'best' solution (line 11), could easily be expanded to include column weights based on which accelerator a subgraph used.

If the $Cover()$ function does not have a complete solution, then two checks are performed to prune the search space before recursing down the search tree (lines 15 - 18). The first check, lines 15 and 16, simply bounds the search tree when it runs out of subgraphs to examine: essentially when it hits leaves of the tree. The second check bounds when the current solution cannot possibly be better than the best known solution, by computing a lower bound on the partial cover, x. The first portion of line 17, $\sum_{i=1}^{n} x_i$, calculates how many subgraphs are in the current cover. The second portion of the equation calculates the number of ops that still need to be covered and divides by the number of ops covered by the current subgraph. Since the subgraphs are sorted by size, and they are always added in order of decreasing size, the second portion of the equation gives a lower bound on the number of additional subgraphs that must be added to complete a cover. The check in line 17 is the primary catalyst that makes this unate covering algorithm practical for subgraph selection.

Improvement Over Previous Work: As with the isomorphism algorithm, there are several techniques that make this unate covering algorithm faster than previous solutions. The first of these is sorting the subgraphs in order of decreasing size (line 3 of Algorithm 5.2). While this does not directly prune the search tree, it does enable other pruning techniques, such as the check in line 17. Another technique is to always branch toward adding a subgraph first (lines 4 and 19). Since the subgraphs are sorted by size, and the subgraphs are considered in consecutive order, always adding ensures the first complete cover will be exactly the same as the greedy solution. The greedy solution provides an excellent bound to quickly prune bad portions of the search tree. Additionally, by reaching the greedy solution first, if the algorithm runs for an unusually long time, it can always be stopped at without fear of a solution worse than greedy.

Unate Covering Example: Figure 5.3 A shows an example of the Boolean matrix, M, used in Algorithm 5.2. This matrix shows several subgraphs which were enumerated from



	A	B	C	...	D	E	F	G	H	I	...	J
1		1					1		1			
2					1		1			1		
3	1	1										
4					1		1					
5					1							
6	1	1										
8	1	1										
9	1	1	1									
10	1	1										
11	1	1	1									
12	1	1	1									
13	1		1			1						
14	1	1	1									
15			1			1		1				
16			1			1		1				1

Figure 5.3: A. Example unate covering problem used to map subgraphs from the basic block in Figure 5.1. B. The mapping solution with full-enumeration and greedy selection. C. Mapping solution with full-enumeration and unate covering selection.

the basic block from g721encode, shown in Figure 5.1 B (many subgraphs were omitted for space and clarity reasons). The subgraphs correspond to an accelerator which has 4 inputs, 2 outputs, and can support any computation with a dependence chain of 4 or less, also pictured in Figure 5.1. Notice how the subgraphs are sorted from largest at the left (covering 9 operations), to the smallest at right (each operation node as a subgraph).

Algorithm 5.2 begins the $Cover()$ function by adding subgraph A, the largest subgraph, to its current cover, x. It will then recurse, and attempt to add B to x. The check at line 7 will prevent this since the two subgraphs overlap, and this branch of the search space will be pruned. Eventually, by moving across the matrix in Figure 5.3, subgraphs D, then G, and then H will be added to A to create a complete cover, shown in Figure 5.3 B. This is the full-enumeration / greedy-selection solution. Assuming a single-issue processor, and the accelerator and each operation in Figure 5.1 B takes one cycle to execute, this solution will yield a speedup of $\frac{17}{3+3} = 2.83$ for this block. The first 3 in the denominator accounts for the right-shift, branch and the multiply that were not accelerated, and the second 3 is for each of the 3 subgraphs that will be run on the accelerator.

After the unate covering algorithm finds the greedy-selection solution, it will continue to explore the search tree and eventually discover the cover B, D, E, shown in Figure 5.3 C. This solution uses fewer subgraphs, and will be recorded as the best solution on line 13. The speedup for this solution is $\frac{17}{2+3} = 3.4$. This compares quite favorably with the speedup obtained using the greedy enumeration - immediate selection described in Section 5.3.1, which is only $\frac{17}{7} = 2.43$. Clearly full-enumeration with unate-covering based selection can provide benefits beyond greedy heuristics.

5.3.2.4 Algorithm Runtimes

There are clearly performance benefits to be had over the standard greedy algorithms, if accelerators can be targeted using the NP-Complete formulations that we have proposed. The major concern is that the proposed algorithms are tractable. Figure 5.4 demonstrates that they are.

Each point in these graphs represents the algorithm runtime of a basic block from 1 of 23 MediaBench [75] and MiBench [53] applications. The data was collected on a 3.06 GHz

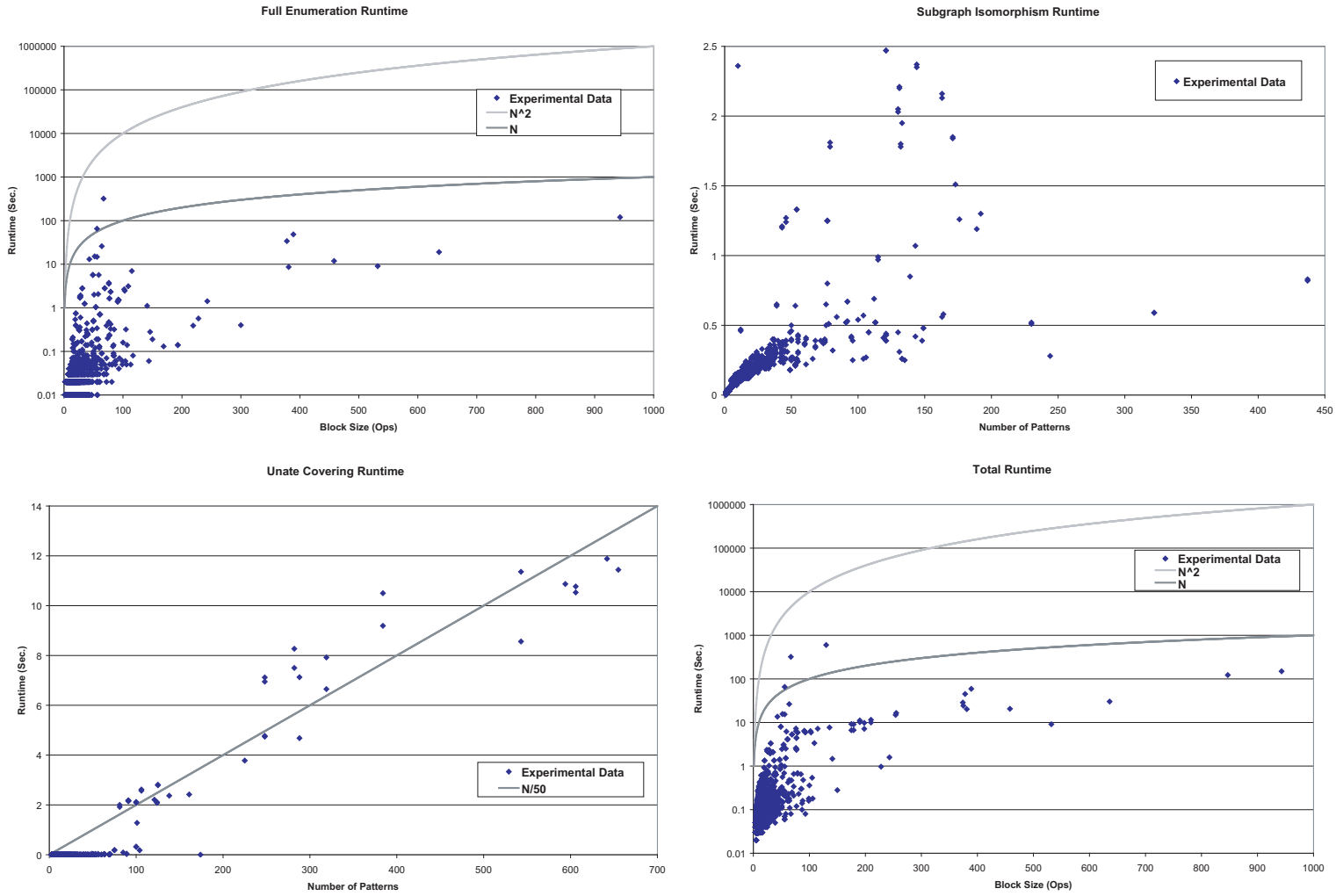


Figure 5.4: Compilation runtimes for various aspects of the proposed algorithms

Pentium 4 machine with 1 GB of RAM. Applications were compiled to target an accelerator with 4 inputs, 2 outputs, and a maximum dependence height of 4 (similar to the accelerator proposed in [30]). Each algorithm was given a maximum time limit of 600 seconds per block, at which point the algorithm was terminated and reported the best solution seen up to that point. Note that *only one basic block out of 23 applications* reached the time limit for any of the proposed algorithms; that was during subgraph enumeration.

To summarize the results for subgraph enumeration, more than 99.8% of basic blocks were fully enumerated in less than 1 second, and more than 99.95% of the blocks were enumerated within 10 seconds. As mentioned previously, the worst case block timed out at 600 seconds. This could be prevented by more aggressively partitioning the block into smaller components. Overall, the enumeration algorithm runtime appeared to grow only *linearly* with the size of the basic block, which makes this algorithm quite scalable.

Runtimes for the subgraph isomorphism algorithm were also very reasonable. More than 99.7% of blocks had subgraph isomorphism checked for all their enumerated subgraphs in less than 1 second. The worst case runtime for any of the blocks was only 2.47 seconds.

As with subgraph enumeration, runtime for unate covering grew roughly linearly with the size of its input matrix, and the runtime was very fast in the common case. More than 99.1% of blocks ran unate covering selection in less than 1 second, while 99.8% finished in less than 10 seconds. The worst case runtime for any block was 60.25 seconds (this was the same block that timed out during enumeration).

In terms of total runtime for all three phases (full enumeration, isomorphism based pruning, and unate covering selection), more than 98% of blocks took less than 1 second to run. 99.5% of basic blocks took less than 10 seconds total. The worst case block out of the 23 applications took 11.03 minutes. If the worst case block proved too slow, the algorithms were designed so that the timeout could easily be reduced without drastically affecting solution quality.

These results show that if you are compiling to target an accelerator statically, runtime is no reason to use a greedy heuristic.

5.4 Experiments

In order to evaluate the proposed mapping algorithm, an experimental framework was built using the Trimaran research compiler [121] and SimpleScalar ARM simulator [9]. Trimaran was retargeted for the ARM instruction set and subgraphs to be accelerated were delineated in the in the binary. After compilation, the simulator recognized the subgraphs and modeled them as if an accelerator was present. SimpleScalar was configured to represent an ARM-926EJ [5], a popular embedded core, with accelerators that took one cycle to execute.

Twenty three benchmarks from MediaBench [75] and MiBench [53] were used to evaluate the proposed mapping algorithms. Omitted benchmarks were due to issues in the compiler infrastructure. We tested three different algorithms: greedy enumeration - immediate selection (as described in Section 5.3.1), full enumeration - unate covering selection, or FEU (described in 5.3.2), and a hybrid technique full enumeration - greedy selection, or FEG.

Algorithm Comparison: Figure 5.5 shows the speedups attained when using the three proposed algorithms to target the 4 input / 2 output accelerator shown in Figure 5.1 A. The figure illustrates that the FEU algorithm consistently outperforms greedy on nearly every benchmark. On average, 9% more speedup was achieved by using the FEU algorithm instead of greedy heuristics. Sha showed the largest difference between greedy and FEU, at 32% improvement. The primary reason for this is that full enumeration identified a considerable number of disconnected subgraphs in the critical loop, which the greedy algorithm was not capable of finding. Dijkstra_large showed the least improvement when moving from greedy to FEU mapping. The important subgraphs in this benchmark only consist of 2 back-to-back instructions, thus the subgraphs are easy to identify regardless of enumeration algorithm. As would be expected, this shows that computation-bound applications with very large basic blocks benefit more from the FEU algorithm than applications with small basic blocks.

One surprising result illustrated in Figure 5.5 is that most applications did not benefit from unate covering selection (comparing FEG with FEU). On average FEU performed

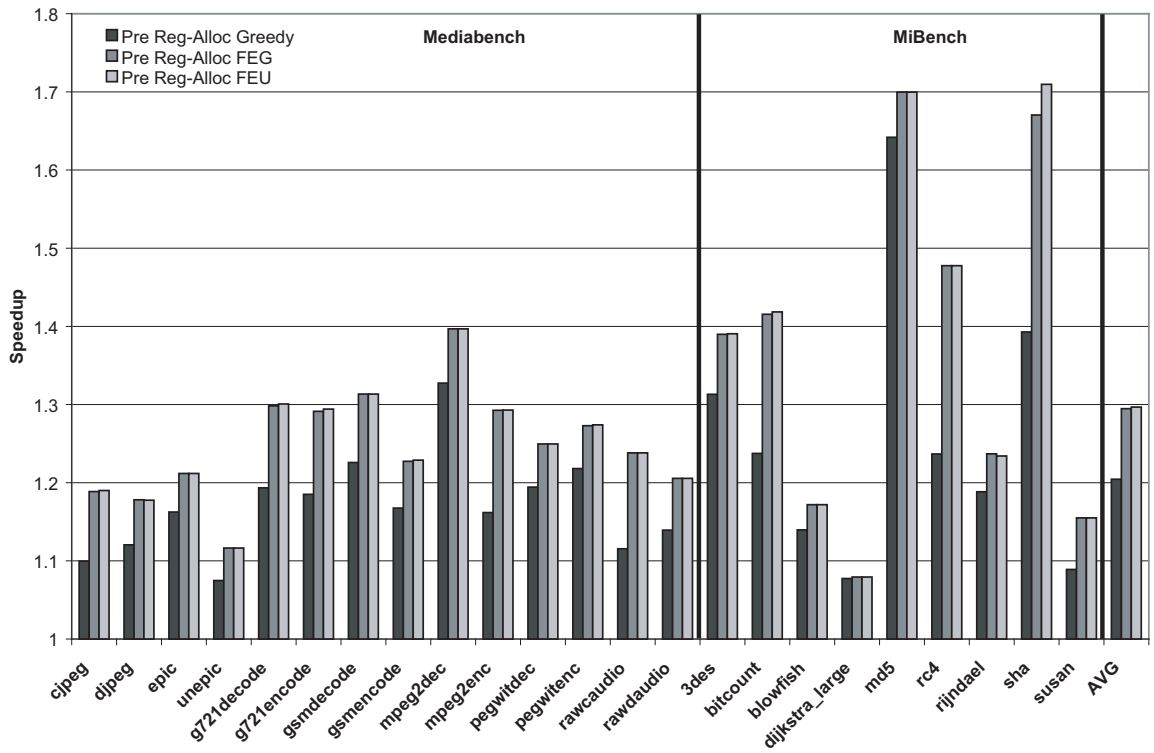


Figure 5.5: Comparison of subgraph mapping algorithms

only 1% better than FEG. The main reason for this is that the critical computation in most basic blocks was small enough that very few subgraphs were needed in the cover. If more subgraphs are used to cover the DFG (for example, when targeting a smaller accelerator) then greedy selection is more likely to get stuck in a local minima and perform worse. However, when targeting the large accelerator from Figure 5.1 A, greedy selection is sufficient. In two instances, djpeg and rijndael, unate covering selection actually caused slight performance decreases. This is due to second-order effects, such as cache alignment, that are not modeled by the unate covering formulation.

Sensitivity to Targeted Accelerator: Figure 5.6 shows how much better FEU performs relative to greedy when varying the targeted accelerator. Bars greater than one imply FEU performed better than greedy and bars less than one imply greedy performed better than FEU. The rightmost bar for each benchmark represents the 4 input / 2 output accelerator used throughout this chapter. The 3 input / 1 output accelerator consists of two back-to-

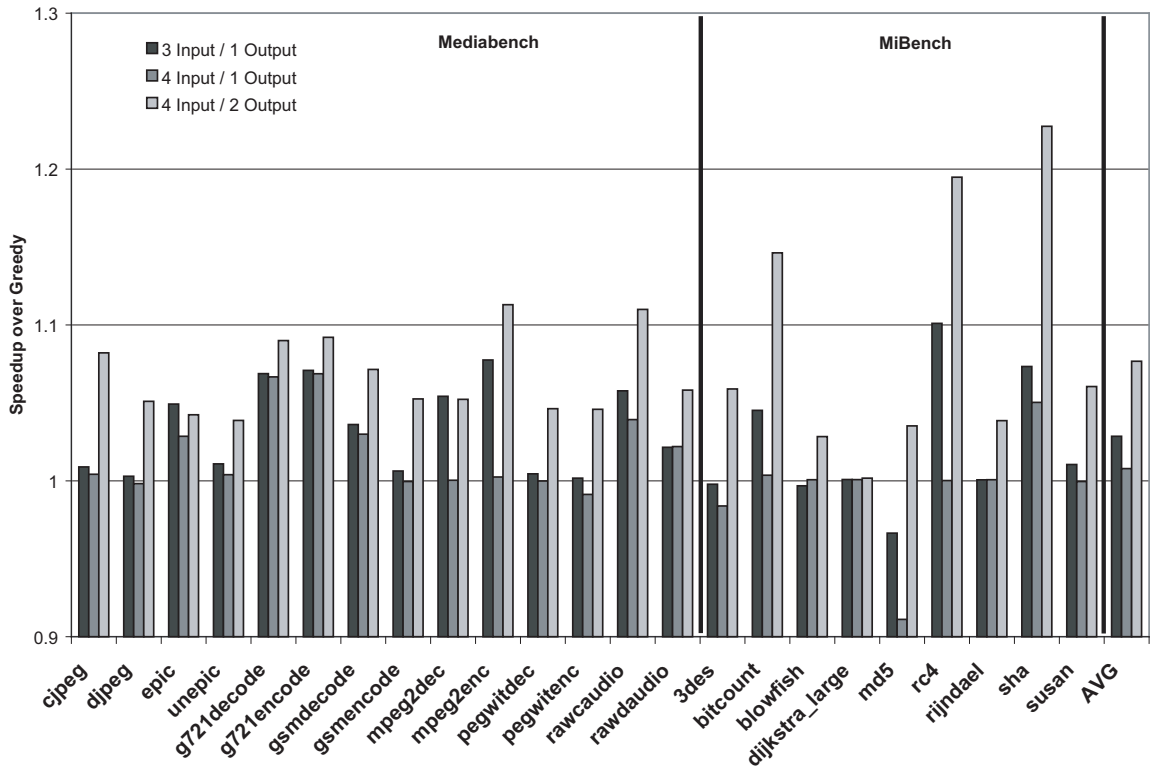


Figure 5.6: The speedup of Full-Enumeration/Unate Covering Selection over Greedy while varying the targeted accelerator

back functions units, and is modeled after the accelerator used in [60]. The 4 input / 1 output accelerator has computation capabilities somewhere in between those two, with 7 function units and maximum dependence height of 3.

There are several interesting trends illustrated by this figure. First note that FEU outperforms greedy much more on the 4/2 configuration than on the 4/1 or the 3/1. This is because accelerators with only one output preclude disconnected subgraphs from being executed. If no disconnected subgraphs are allowed, then greedy can potentially find the same subgraphs as full enumeration. This definitely helps narrow the gap between the two algorithms. In general, larger accelerators with multiple inputs and outputs place more importance on high quality subgraph enumeration.

A second important trend in Figure 5.6 is that FEU outperforms greedy more in 3/1 than in 4/1. The reason for this is that the small number of function units in 3/1 (only 2)

made the number of subgraphs selected in the final cover relatively high, compared with 4/1 which has 7 function units. Since more subgraphs are needed, more emphasis is placed on the covering algorithm, and unate covering helped quite a bit. The 4/1 accelerator used relatively few subgraphs, that were all discoverable via greedy enumeration, therefore FEU provided little benefit beyond the greedy algorithm. This shows that more thorough strategies, used in FEU, are more important whenever the search space is very large.

A last trend to note in Figure 5.6 is that in certain benchmarks, such as md5, greedy actually performed better than FEU. This is due to the partitioning used during full enumeration. Recall that in order to make full enumeration tractable, very large blocks have to be partitioned into smaller pieces. Occasionally this partitioning precludes full enumeration from finding important subgraphs which can be discovered by greedy methods. This problem is pronounced in accelerators with only one output, since full enumeration cannot make up ground on greedy by using disconnected subgraphs. Figure 5.6 motivates future work to develop faster enumeration algorithms and better partitioners to alleviate the problem in md5.

Effect of Register Allocation: Figure 5.7 depicts the result of applying the FEU mapping algorithm before and after register allocation. This is an important result because many researchers have proposed subgraph mapping in virtual machines or as a part of binary-to-binary translation. The drawback of subgraph mapping after register allocation is that spill code essentially breaks dataflow edges by placing values in memory. This limits the size of computation subgraphs that can be identified for acceleration. On the other hand, register allocation does introduce some additional computation (e.g. stack adjustments) that could potentially be accelerated, which is not available when mapping before allocation.

On average, we found that performing subgraph mapping prior to allocation produced results with 8% more speedup than post-allocation mapping. In some benchmarks, like rawcaudio, the innermost loop was so small that there was no spill code, and so there was no difference in the results. In other benchmarks, such as 3des, the amount of spill code was so large that virtually none of the pre-allocation subgraphs were discoverable post-allocation. Only one benchmark, epic, performed better from post-allocation mapping. Figure 5.7 clearly shows that performing subgraph mapping pre-allocation in the compiler

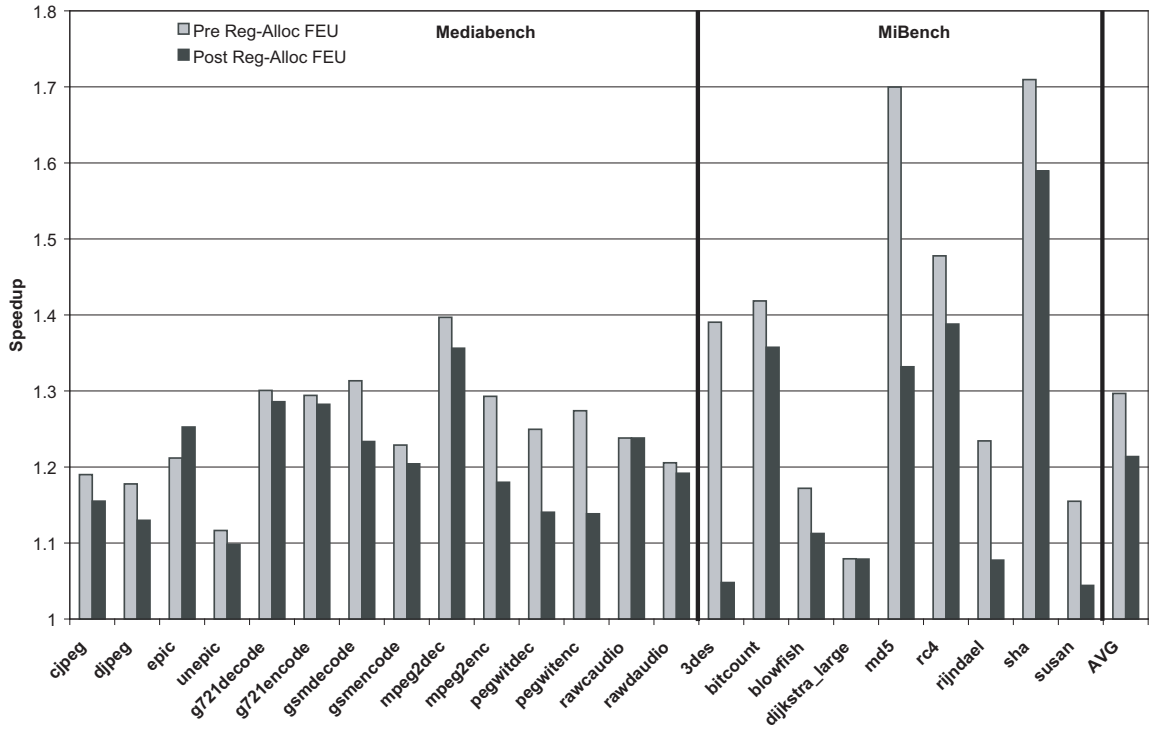


Figure 5.7: Comparison of mapping effectiveness before and after register allocation using the accelerator from Figure 5.1 A

is much more effective than post compilation techniques, such as binary translation.

5.5 Summary

In this chapter, we addressed the inefficiencies of traditional compiler algorithms used to identify candidate subgraphs for execution on computation accelerators. Several new algorithms were developed to find better candidates for both small and larger acyclic accelerators. Simulation results demonstrate that our proposed algorithms achieve, on average 9%, and as much as 32% more speedup than traditional greedy solutions.

This work also quantified the effect of register allocation on subgraph identification. On average, performing subgraph mapping prior to register allocation results in 8% more speedup. This result implies that performing dynamic subgraph identification in hardware or a virtual machine would significantly reduce the effectiveness of mapping algorithms.

CHAPTER 6

Applying Transparent Customization to SIMD Accelerators

6.1 Introduction

Single-instruction multiple-data (SIMD) accelerators are commonly used in microprocessors to accelerate the execution of media applications. These accelerators perform the same computation on multiple data items using a single instruction. To utilize these accelerators, the baseline instruction set of a processor is extended with a set of SIMD instructions to invoke the hardware. Intel's MMX and SSE extensions are examples of two generations of such instructions for the x86 instruction set architecture (ISA). SIMD accelerators are popular across desktop and embedded processor families, providing large performance gains at low cost and energy overheads.

As with the acyclic accelerators discussed in previous chapters, SIMD accelerators are a proven mechanism to improve performance. However, the forward migration path from generation to generation is a difficult problem. SIMD hardware evolves in terms of width and functionality with each generation. For example, the Intel MMX instructions operated on 64-bit vectors and this was expanded to 128-bit for SSE2. The opcode repertoire is also commonly enhanced from generation to generation to account for new functionality present in the latest applications. For example, the number of opcodes in the ARM SIMD instruction set went from 60 to more than 120 in the change from Version 6 to 7 of the ISA.

SIMD evolution in desktop processors has been relatively stagnant recently, with vector lengths standardizing at 4 to 8 elements. However, this is not the case in embedded systems. For example, the ARM Neon SIMD instructions were extended from 4 to 16 8-bit elements in 2004 [12]. Other recent research [81] has proposed vector lengths of 32 elements are the most suitable size for signal processing accelerators. Undoubtedly, SIMD architectures are still evolving in many domains.

Migration to new generations of SIMD accelerators is very difficult, though. Once an application is targeted for one set of SIMD instructions, it must be rewritten for the new set. Hand-coded assembly is commonly used to exploit SIMD accelerators; thus, rewriting applications is time consuming, error prone, and tedious. Programming with a library of intrinsics can mitigate the problem to some degree, but software migration still requires substantial effort, as code is usually written assuming a fixed SIMD width and functionality.

To effectively deal with multiple generations of SIMD accelerators and overcome the software migration problems, this chapter investigates the use of delayed binding with SIMD accelerators. Delayed binding is a technique used in many areas of computer science to improve the flexibility and the efficiency of systems. For example, dynamic linkers delay the binding of object code to improve portability and space efficiency of applications; dynamic compilers take advantage of late binding to perform optimizations that would otherwise be difficult or impossible without exact knowledge of a program's runtime environment [51]. Examples of delayed binding in processors include the use of trace caches and various techniques for virtualization [99]. Just as in software systems, these techniques aim to improve flexibility and efficiency of programs, but often require non-trivial amounts of hardware and complexity to deploy.

Similar to the transparent instruction set customization presented in Chapter 4, delayed binding of SIMD accelerators is accomplished through compiler support and a translation system, collectively referred to as *Liquid SIMD*. The objective is to separate the SIMD accelerator implementation from the ISA, providing an abstraction to overcome ISA migration problems. Compiler support in Liquid SIMD translates SIMD instructions into a virtualized representation using the processor's baseline instruction set. The compiler also isolates portions of the application's dataflow graph to facilitate translation. The

translator dynamically identifies these isolated dataflow subgraphs, and converts them into architecture-specific SIMD instructions.

Liquid SIMD offers a number of important advantages for families of processor implementations. First, SIMD accelerators can be deployed without having to alter the instruction set and introduce ISA compatibility problems. These problems are prohibitively expensive for many practical purposes. Second, delayed binding allows an application to be developed for one accelerator, but be utilized by completely different accelerators (e.g., an older or newer generation SIMD accelerator). This eases non-recurring engineering costs in evolving SIMD accelerators or enables companies to differentiate processors based on acceleration capabilities provided. Finally, SIMDized code in a Liquid SIMD system can be run on processors with no SIMD accelerator or translator, simply by using native scalar instructions.

The contributions of this chapter are fourfold:

- It describes an compiler/translation framework to realize Liquid SIMD, which decouples the SIMD hardware implementation from the ISA.
- It develops a simple, ISA-independent mechanism to express width-independent SIMDization opportunities to a translator.
- It presents the design and implementation of a light-weight dynamic translator capable of generating SIMD code at runtime.
- It evaluates the effectiveness of Liquid SIMD in terms of exploiting varying SIMD accelerators, the runtime overhead of SIMD translation, and the costs incurred from dynamic translation.

6.2 Overview of the Approach

SIMD accelerators have become ubiquitous in modern general purpose processors. MMX, SSE, 3DNow!, and AltiVec are all examples of instruction set extensions that are tightly coupled with specialized processing units to exploit data parallelism. A SIMD ac-

celerator is typically implemented as a hardware coprocessor composed of a set of functional units and an independent set of registers connected to the processor through memory. SIMD accelerator architectures vary based on the width of the vector data along with the number and type of available functional units. This allows for diversity in two dimensions: the number of data elements that may be operated on simultaneously and the set of available operations.

The purpose of this chapter is to decouple the instruction set from the SIMD accelerator hardware by expressing SIMD optimization opportunities using the processor's baseline instruction set. Expressing SIMD instructions using the baseline instruction set provides an abstract software interface for the SIMD accelerators, which can be utilized through a lightweight dynamic translator. This lessens the development costs of the SIMD accelerators and provides binary compatibility across hardware and software generations.

There are two phases necessary in decoupling SIMD accelerators from the processor's instruction set. First, an offline phase takes SIMD instructions and maps them to an equivalent representation. Second, a dynamic translation phase turns the scalar representation back into architecture-specific SIMD equivalents.

Converting SIMD instructions into an equivalent scalar representation requires a set of rules that describe the conversion process, analogous to the syntax of a programming language. The conversion can either be done at compile time or by using a post-compilation cross compiler. It is important to note that the SIMD-to-scalar conversion is completely orthogonal to automated SIMDization (i.e., conversion can be done in conjunction with compiler-automated SIMD code or with hand coded assembly). Further, no information is lost during this conversion. The resulting scalar code is functionally equivalent to the input SIMD code, and a dynamic translator is able to recover the SIMD version provided it understands the conversion rules used.

Dynamic translation converts the virtualized SIMD code (i.e., the scalar representation) into processor-specific SIMD instructions. This can be accomplished using binary translation, just-in-time compilation (JITs), or hardware. Offline binary translation is undesirable for three reasons. First, there is a lack of transparency; user or OS intervention is needed to translate the binary. Second, it requires multiple copies of the binary to be kept. Lastly,

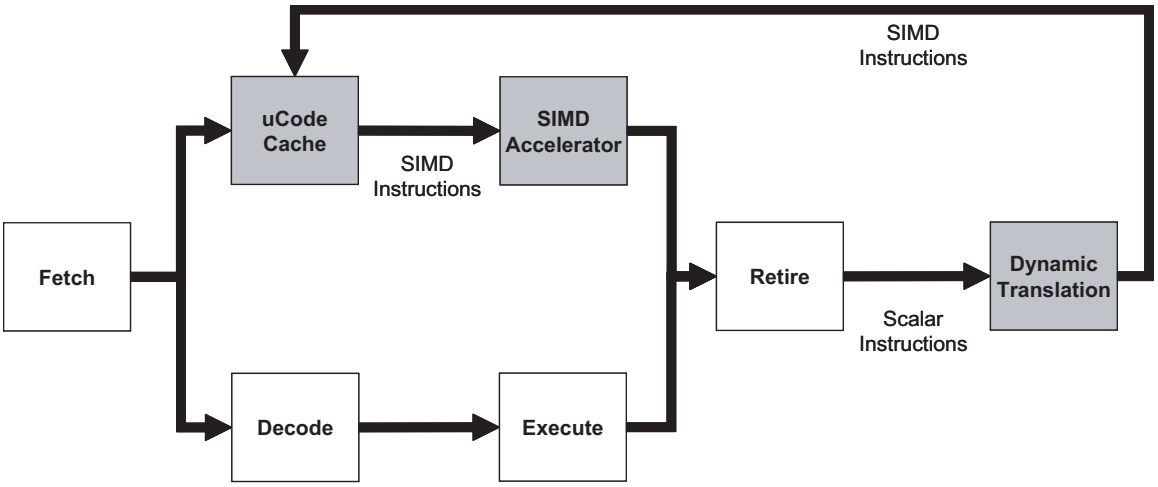


Figure 6.1: Pipeline organization for Liquid SIMD. Gray boxes represent additions to a basic pipeline.

there is an accountability issue when applications break. Is the application developer or the translator at fault?

JITs or virtual machines are more viable options for dynamic translation. However, in this chapter we present the design of a dynamic translator using hardware. The main benefit of hardware-based translation over JITs is that it is more efficient than software approaches. This chapter shows that the translation hardware is off the processor's critical path and takes less than 0.2 mm^2 of die area. Additionally, hardware translation does not require a separate translation process to share the CPU, which may be unacceptable in embedded systems. Nothing about our virtualization technique precludes software-based translation, though.

The remainder of this chapter describes a compiler technique for generating code for an abstracted SIMD interface, coupled with a post-retirement hardware method for dynamic translation. Our high level processor architecture is presented in Figure 6.1. A basic pipeline is augmented with a SIMD accelerator, post-retirement dynamic translator, and a microcode cache that stores recently translated SIMD instructions. This system provides high-performance for data parallel operations without requiring instruction set modifications or sacrificing binary compatibility.

6.3 Liquid SIMD Compilation

The purpose of the compiler in the Liquid SIMD framework is to translate SIMD instructions into an equivalent scalar representation. That is, the compiler re-expresses SIMD instructions using an equivalent set of instructions from the processor's scalar ISA. Since the scalar ISA is Turing-complete, any SIMD instruction can be represented using the scalar ISA. The challenge is finding a representation that is easy to convert back to SIMD and is also relatively efficient in its scalar form.

It is important note that this chapter is not proposing any techniques that rely on the compiler to automatically SIMDize a program. While the approach presented could be used in conjunction with automatic SIMDization techniques [14, 40, 69, 73, 127], this is not the main focus of this chapter. Instead, we focus on how to design a scalar representation of SIMD code, which executes correctly on a baseline processor, and is amenable to runtime translation.

6.3.1 Hardware and Software Assumptions

Before describing the actual strategy for abstraction, it is important to explicitly state some assumptions about the hardware targeted and applications to be run. First, it is assumed that the targeted SIMD accelerators operate as a separate pipeline. That is, the SIMD accelerator shares an instruction stream and front end with a baseline pipeline, but has separate register files and execution units.

Second, it is assumed that the SIMD accelerator uses a memory-to-memory interface. That is, when executing SIMD instructions, the basic sequence of events is a loop that loads vectors, operates on them, and finally stores the vectors back to memory. In this model, there is no register-to-register communication between the scalar register file and the vector register file, and intermediate data not stored to memory is not accessed by successive loops. The assumption that there is little register-to-register communication is validated by production SIMD accelerators, which usually have either very slow or no direct communication between the two register files. The lack of intermediate data communication between loops is a side-effect of the types of loops being optimized; typically the ideal

size of a vector, from the software perspective, is much too large to fit into the hardware vector size. For example, one of the hot loops in 171.swim operates on vectors of size 514. If hardware supported vectors that long, then computed results could be passed between successive loops in a register. Since the results do not fit in hardware, the results have to be passed through memory.

A last assumption is that the application must be compiled to some maximum vectorizable length. That is, even though the binary will be dynamically adjusted based on the vector width supported in the hardware, there is some maximum vector width supported by the binary. The reason for this assumption is due to memory alignment. Most SIMD systems restrict memory accesses to be aligned based on their vector length. To enforce such alignment restrictions, the compiler aligns data based on an assumed maximum width. The binary can be dynamically adjusted to target any width less than the maximum. The trade off here is code size may unnecessarily increase if an accelerator supports narrower widths than the assumed vector size.

Implicit in this alignment restriction is the assumption that targeted accelerators only support execution widths that are a power of 2 (i.e., 2, 4, 8, ...). That is, a binary compiled for maximum vector width of 8 could not (easily) be dynamically translated to run on a 3-wide SIMD accelerator, because data would be aligned at 8 element boundaries in the binary. Assuming SIMD accelerators are power-of-2 widths is certainly valid for the majority of SIMD accelerators in use today.

6.3.2 Scalar Representation of SIMD Operations

With these assumptions in mind, we now discuss how to convert SIMD instructions into an equivalent scalar representation. The conversion rules are shown in Table 6.1. This section will walk through the thinking behind these rules, and Section 6.3.4 will demonstrate the usage of the rules in a detailed example.

The most natural way to express SIMD operations using scalar instructions is by creating a scalar loop that processes one element of the SIMD vector per iteration. Since SIMD accelerators have a memory-memory interface, vector loads can be converted to scalar

SIMD Category	Example SIMD Instruction	Scalar Equivalent	Comments
(1) Data parallel; operates on two vectors	`v1 = vadd v2, v3`	`r1 = add r2, r3`	Used for any operation which has an equivalent scalar operation. SIMD operations without a scalar equivalent (e.g., saturating arithmetic) must construct an idiom using multiple instructions.
(2) Data parallel; operates on a vector and a scalar supported constant	`v1 = vand v2, 0xFF`	`r1 = and r1, 0xFF`	Analogous to category (1)
(3) Data parallel; operates on vector and non-scalar supported constant	`v1 = vor v2, 0xFF00FF00`	`r3 = ld [cnst + ind]` `r1 = or r2, r3`	Compiler inserts a read-only array, `cnst`, into the code, which stores the unsupported constant. The array is indexed using the loop's induction variable to retrieve the appropriate portions during each scalar iteration.
(4) Reductions; multiple vector elements used to compute one result	`r1 = vmin v2`	`r1 = min r1, r2`	Loop-carried dependence (`r1`) is used to represent that each element of the vector is used to calculate one result.
(5) Memory accesses	`v1 = vldb [addr]`	`r1 = ldb [addr + ind]`	Induction variable is used to select one vector element to operate on each iteration. Loads are used to identify width of vector elements (e.g., byte or halfword).
(6) Base-plus-displacement memory accesses	`[addr + r1] = vstr v2`	`r3 = add r1, ind` `[addr + r3] = str r2`	Similar to category (5),

(7) Permutations; reorders vector elements	`v2 = vld [addr]`
	`v1 = vbfly v2`
		`r3 = ld [bfly + ind]`
		`r4 = add ind, r3`
		`r1 = ld [addr + r4]`
			Compiler inserts a read-only array, `bfly`, into the code, which stores how elements are reordered. This is used in conjunction with the induction variable to bring in vector elements in a different order. Values stored in `bfly` uniquely identify a permutation.

(8) Permutations; reorders vector elements	`v1 = vbfly v2`
	`[addr] = vstr v1`
		`r3 = ld [bfly + ind]`
		`r4 = add ind, r3`
		`[addr + r4] = str r1`
			Analogous to category (7), but writes elements to memory in a different order, instead of reading them.

Table 6.1: Rules for translating SIMD instructions into scalar equivalents. Operands beginning with `r` are scalars, operands beginning with `v` are vectors, and `ind` is the loop's induction variable.

loads using the loop's induction variable to select a vector element. The size of a vector's elements is derived from the type of scalar load used to read the vector (e.g., load-byte means the vector is composed of 8-bit elements). Similar to memory accesses, data parallel SIMD operations can be represented with one or more scalar instructions that perform the same computation on one element of the vector. Essentially, any data parallel SIMD instruction can be converted to scalar code by operating on one element of the SIMD vector at a time.

If any SIMD operation does not have a scalar equivalent (e.g., many SIMD ISAs but few scalar ISAs support saturating arithmetic), then the scalar equivalent can be constructed using an idiom consisting of multiple scalar instructions. For example, 8-bit saturating addition could be expressed in the ARM scalar ISA as `r1 = add r2, r3; cmp r1, 0xFF; movgt r1, 0xFF`, where the move instruction is predicated on the compari-

son. Vector masks, or element-specific predication, is another common example of a SIMD instruction that would likely be constructed using idioms. A dynamic translator can recognize that these sequences of scalar instructions represent one SIMD instruction, and no efficiency is lost in the dynamically translated code. Again, the scalar instruction set is Turing-complete, so any data parallel SIMD instruction *can* be represented using scalar instructions. The only downside is potentially less efficient scalar code if no dynamic translator is present in the system.

More complicated SIMD instructions, which operate on all vector elements to produce one result (e.g., max, min, and sum), can be represented using a loop-carried register in the scalar loop. For example, category (4) in Table 6.1 shows how a vector min can be represented. If the result register is used both as a source and destination operand, and no other operation defines `r1` in the loop, then `r1` will accumulate the minimum of each vector element loaded into `r2`. The dynamic translator can easily keep track of which registers hold loop-carried state, such as `r1` in this example, meaning vector operations that generate a scalar value fit into the Liquid SIMD system.

One difficulty in using a scalar loop representation of SIMD instructions is handling operations that change the order of vector elements. Permutation instructions illustrate this problem well. Suppose a loop is constructed and begins operating on the first element of two SIMD vectors. After several data parallel instructions, a permutation reorders the vector elements. This means that the scalar data that was being operated on in one loop iteration is needed in a different iteration. Likewise, the permutation causes scalar data from future (or past) iterations to be needed in the current iteration.

To overcome this problem, we propose limiting permutation instructions to only occur at memory boundaries of scalar loops. This allows the reordering to occur by using loads or stores with a combination of the induction variable and some statically defined offset. Essentially, this loads the correct element for each iteration.

The last two rows of Table 6.1 briefly illustrate how reordering at memory boundaries works. In category (7), a butterfly instruction reorders the elements of `v2`. In order for the scalar loop to operate on the correct element each iteration, the induction variable needs to be modified by an offset, based on what type of permutation is being performed. The

compiler creates a read-only array, `bfly`, that holds these offsets. Once the offset is added to the induction variable, the scalar load will bring in the appropriate vector element. A dynamic translator uses the offsets to identify what type of permutation instruction is being executed in the scalar equivalent. Offsets are used, as opposed to absolute numbers, to ensure vector width independence of the scalar representation.

The downside of using offsets to represent permutations is that element reordering operations must occur at scalar loop boundaries using a memory-memory interface. This makes the code inherently less efficient than standard SIMD instruction sets, which can perform this operation in registers.

Using only the rules in Table 6.1 and simple idiom extensions, we were able to express the vast majority of the ARM Neon SIMD instruction [12] set using the scalar ARM ISA. Neon is a fairly generic SIMD instruction set, meaning the techniques developed here are certainly applicable to a wide variety of other architectures.

6.3.3 Limitations of the Scalar Representation

Although using this scalar representation has many benefits, there are some drawbacks that must be taken into consideration. The most obvious is that virtualized SIMD code will not be as efficient on scalar processors as code compiled directly for a scalar processor. This is primarily because of the memory-to-memory interface, the lack of loop unrolling, and the use of idioms. Performance overhead is likely to be minimal, though, since vectors in the working set will be cache hits, the loop branch is easy to predict, and the idioms used are likely to be the most efficient scalar implementation of a given computation. Another mitigating factor is that the scalar code can be scheduled at the idiom granularity to make the untranslated code as efficient as possible. As long as the idioms are intact, the dynamic translator will be able to recover the SIMD code.

Another drawback of the proposed virtualization technique is increased register pressure. Register pressure increases because the scalar registers are being used to represent both scalars and vectors in the virtual format. Additionally, temporary registers are needed for some of the proposed idioms. This could potentially cause spill code which degrades

performance of both the scalar representation and translated SIMD code. Empirically speaking, register pressure was not a problem in the benchmarks evaluated in this chapter.

A last limitation is that there are two classes of instructions, from ARM's Neon ISA, which are not handled by the proposed scalar representation. One such instruction is `v1 = VTBL v2, v3`. In the `VTBL` instruction, each element of `v2` contains as an index for an element of `v3` to write into `v1`. For example, if the first element of `v2` was 3, then the third element of `v3` would be written into the first element of `v1`. This is difficult to represent in the proposed scalar representation, because the induction variable offset, which defines what vector elements are needed in the current loop iteration, is not known until runtime. All other permutation instructions in Neon define this offset statically, allowing the compiler to insert a read-only offset array in the code.

The second class of unsupported instructions is interleaved memory accesses. Interleaving provides an efficient way to split one memory access across multiple destination registers, or to write one register value into strided memory locations. This is primarily used to aggregate/disseminate structure fields, which are not consecutive in memory. There is no scalar equivalent for interleaved memory accesses, and equivalent idioms are quite complex.

The performance of certain applications will undoubtedly suffer from not supporting these two classes. None of the benchmarks evaluated utilized these instructions, though, meaning the most important SIMD instructions *are* supported by the proposed scalar representation.

6.3.4 SIMD to Scalar Example

To illustrate the process of translating from SIMD to the scalar representation, this section walks through an example from the Fast Fourier Transformation (FFT) kernel, shown in Figure 6.2. There is a nested loop here, where each iteration of the inner loop operates on eight elements of floating point data stored as arrays in memory. This is graphically illustrated in Figure 6.3. The compiler (or engineer) identifies that these operations are

```
for(i = 0; i < 128; i += 8) {
  for(j = i, n = 0; n < 4; j++, n++) {

    k = j + 4;

    tr = ar[i] * RealOut[k] -
           ai[i] * ImagOut[k];

    RealOut[k] = RealOut[j] - tr;
    RealOut[j] += tr;
  }
}
```

Figure 6.2: Example FFT loop.

suitable for SIMD optimization and generates vector load instructions for each eight element data segment. The compiler then schedules vector operations for the loaded data so that the entire inner loop may be executed as a small sequence of SIMD operations, shown in Figure 6.4(A).

Figure 6.4(B) presents the scalar mapping of the SIMD code from Figures 6.3 and 6.4(A). Here, the vector operations of the SIMD loop are converted into a series of sequential operations, and the increment amount of the induction variable is decreased from eight to one, essentially converting each eight element operation into a single scalar operation. The vector load and butterfly instructions in lines `2-5` of the SIMD code are converted into a set of address calculations and load instructions in lines `2-5` of the scalar code. As previously mentioned, SIMD permutation operations are converted into scalar operations by generating a constant array of offset values added to the loop's induction variable. These offsets are stored in the static data segment of the program at the label `bfly`. The value stored at the address `bfly` plus the induction variable value is the offset of the element of the data array to be loaded in the current iteration.

Most of the vector operations from the SIMD code in lines `6-18` are data parallel, and simply map to their scalar equivalent operation (e.g., the `vmult` on SIMD line 8 is converted to a `mult` on scalar line 8). However, there are a few considerations that need to be made for non-parallel operations. Note that the operation on line `17` of the SIMD code requires that all of the values in `vf3` be computed before the `or` operation, because

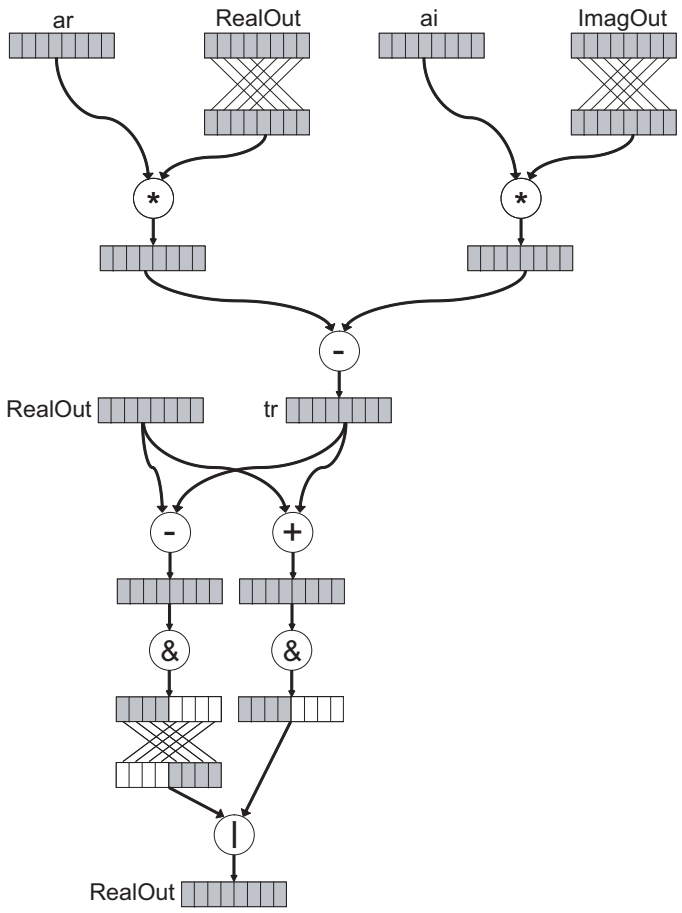


Figure 6.3: Vector representation of Figure 6.2.

the `vbfly` operation in line `15` exchanges the position of the first and last vector element. In order to properly transform this code segment into a set of scalar instructions, the loop body for the scalar code must be terminated early, and the operands to the `or` operation must be calculated and stored in a temporary location at the end of each loop iteration, as shown in lines `18-19` of the scalar code. Then, a second loop is created (lines `24-30`) that performs the serial `or` operation across each element of data. By separating scalar equivalents in different loops, the compiler essentially performs a loop fission optimization to ensure that certain SIMD operations are fully completed before others in the next loop are started.

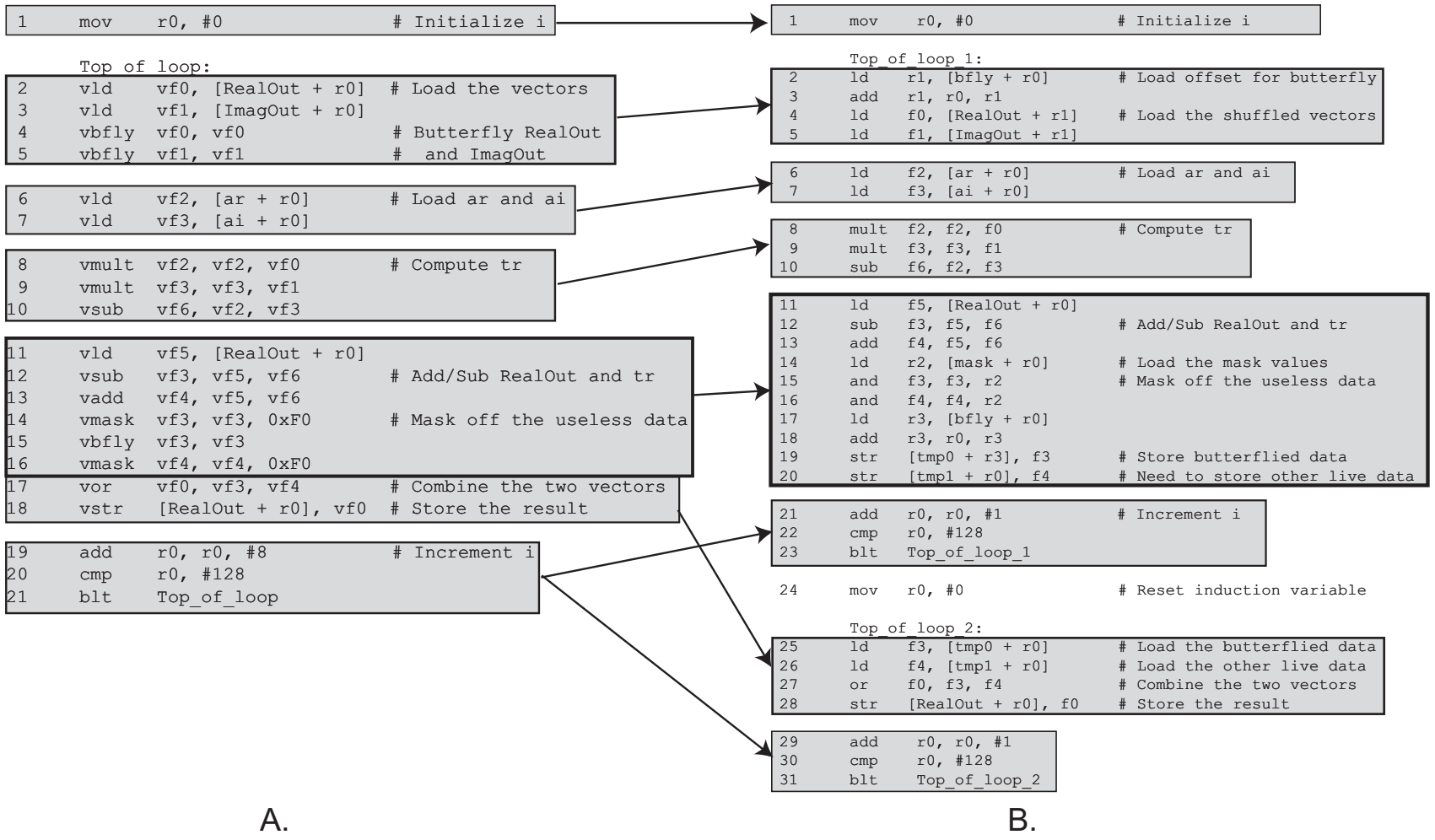


Figure 6.4: (A) SIMD code for Figure 6.2, and (B) scalar representation of the SIMD code in Figure 6.4(A).

6.3.5 Function Outlining

Once the SIMD instructions are translated into scalar code, the compiler needs some way to identify to the translator that these portions of code are translatable. This is accomplished by outlining the code segment as a function, similar to the technique proposed in [30]. The scalar equivalent code is surrounded by a branch-and-link and a return instruction so that the dynamic translator is notified that a particular region of code has potential for SIMD optimization.

In the proposed hardware-based translation scheme, when a scalar region is translated into SIMD instructions, the SIMD code is stored in the microcode cache (see Figure 6.1), and the branch-and-link is marked in a table in the processor's front end. The next time this branch is encountered, the front end can utilize the SIMD accelerator by simply accessing the SIMD instructions in the microcode cache and ignoring the branch. This allows a processor to take advantage of SIMD accelerators without explicit instruction set modifications.

One potential problem with marking translatable code regions by function calls is false positives. This happens if the dynamic translator creates SIMD code for a function that was not meant to be SIMDized. Typically, this is not a problem. ABIs require that functions have a very specific format, which does not match the outlined function format described for scalarized loops. Therefore, the dynamic translator would not be able to convert most non-translatable functions. Even if the translator was able to convert a function that it was not meant to, the SIMD code would be functionally correct as long as there were no memory dependences between scalar loop iterations. Remember, the translator is simply converting between functionally equivalent representations. The scenario of a false positive that produces incorrect code is highly unlikely, but the only way to guarantee correctness is to mark the outlined functions in some unique way (e.g., a new branch-and-link instruction that is only used for translatable regions).

6.4 Dynamic Translation to SIMD Instructions

Once a software abstraction is defined for describing SIMD instructions using a scalar ISA, there needs to be a runtime method for translating them back into SIMD instructions. As mentioned in Section 6.2, there are many valid ways to do this: in hardware at decode time, in hardware after instruction retirement, or through virtual machines or JITs. The software abstraction presented in the previous section is independent of the translation scheme.

Here, the design of a post-retirement hardware translator is presented. Hardware was chosen because the implementation is simple, it adds little overhead to the baseline processor, and hardware is more efficient than software. Post-retirement hardware was chosen, instead of decode time, because post-retirement is far off the critical path of the processor. Our experiments in Section 6.5 and previous work [45] both show that post-retirement optimizations can be hundreds of cycles long without significantly affecting performance. The biggest downside to a post-retirement dynamic mapping is that the modified microcode needs to be stored in a cache and inserted into the control stream in the pipeline frontend.

6.4.1 Dynamic Translation Hardware

From a high level, the translator is essentially *a hardware realization of a deterministic finite automaton that recognizes patterns of scalar instructions to be transformed into SIMD equivalents.* Developing automata (or state machines) to recognize patterns, such as the patterns in Table 6.1, is a mature area of compiler research. A thorough discussion of how to construct such an automata is described in [2].

The structure of the proposed post-retirement dynamic translator is shown in Figure 6.5. To prove the practicality of this structure, it was implemented in HDL (targeting the ARM ISA with Neon SIMD extensions) and synthesized using a 90 nm IBM standard cell process. The results of the synthesis are shown in Table 6.2. Notice that the control generator runs at over 650 MHz, and takes up only 174,000 cells (less than 0.2 mm^2 in 90 nm), without using any custom logic. This shows that the hardware impact of the control generator is well within the reach of many modern architectures.

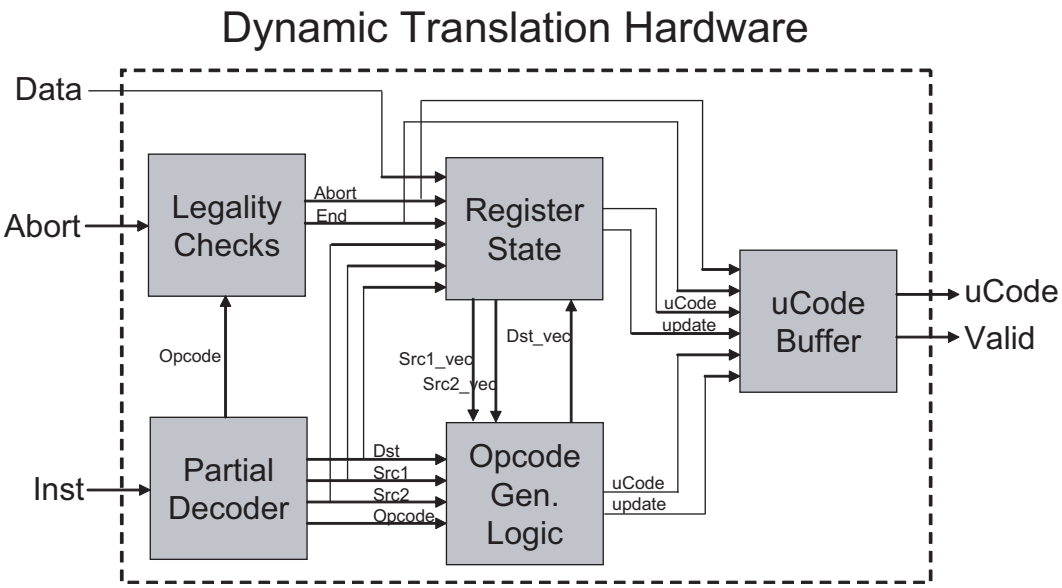


Figure 6.5: Structure of the proposed translator.

Description	Crit. Path	Delay	Area
8-wide Translator	16 gates	1.51 ns	174,117 cells

Table 6.2: Synthesis results for the dynamic translator.

Partial Decoder: The dynamic translator has three inputs from retirement of the baseline pipeline: the instruction that retires (`Inst` in the figure), the data value that instruction generated (`Data`), and an abort signal (`Abort`). Initially, the retired instruction is fed into a partial decoder to determine the source/destination operands and the opcode. It is only a partial decoder, because it only needs to recognize opcodes that are translatable; any other opcodes simply cause translation to abort mapping of the outlined function. This portion of the control generator is potentially redundant, depending on the microarchitecture, because the retiring instruction will likely have the opcode and operand information stored in its pipeline latch. Overall, the partial decoder only takes a few thousand cells of die area, so it does not contribute significantly to the area overhead; it is responsible for 5 of the 16 gates in the critical path, though.

Legality Checks: The purpose of the legality checker in the dynamic translator is to monitor the incoming instructions to ensure that they can be translated. Scalar instructions that do not map to a SIMD equivalent generate an abort signal that flushes stateful portions of dynamic translator. In addition to an instruction generated abort signal, there is an abort signal from the base pipeline to stop translation in the event of a context switch or other interrupt. The legality checker also signals when a subgraph has finished mapping, enabling the microcode buffer to write the translated SIMD instructions into the microcode cache. The legality checks only comprise a few hundred cells and do not occur on the critical path.

Register State: After the instruction is decoded, the operands/opcode access some state, which is indexed based on the register numbers. This register state determines the translation strategy for this instruction. Register state also includes whether or not a register represents a scalar or vector, the size of the data currently assigned to the register (e.g., 16 or 32 bit), and previous values stored in the register. The opcode and register state comprise the data used to transition between states in the automata.

Overall, there are 56 bits of state per register and a large number of MUXes in the register state module, making this structure comprise 55% of the control generator die area. Since the ARM ISA only has 16 architectural integer registers, 55% of the die area is likely proportionally smaller than dynamic translators targeting architectures with more registers. Additionally, this structure will increase in area linearly with the vector lengths of the targeted accelerator.

The previous values assigned to each register are stored in the register state in order to identify operations that are defined using offsets in memory (e.g., the butterfly instruction discussed in Section 6.3). Recall that instructions that reorder elements within a vector are encoded by loading an offset vector, adding the offsets to the induction variable, and using that result for a memory access. In the dynamic translator, load instructions cause the data to be written to the destination register's state. When a data processing instruction uses that destination register as a source operand, (e.g., to add those values to the induction variable), then the previous values of the address are copied to the data processing instruction's destination register state. When a memory access instruction uses a source that has previous values recorded in the register state, this signals that a shuffle may be

Scalar Instruction	Current Register State	Updated Register State	Instruction(s) Generated
(1) r1 = mov #const		r1 is marked as the induction variable	r1 = mov #const
(2) r1 = ld [r2 + r3]	r2 is a scalar; r3 is the induction variable	r1 is a vector; size of r1 is recorded (i.e., byte, halfword, etc.); value loaded is stored in r1	v1 = vld [r2 + r3]
(3) r1 = ld [r2 + r3]	r2 is a scalar; r3 is a vector; r3 has values loaded into it from an offset array	r1 is a vector; size of r1 is recorded	v1 = vld [r2 + ind] v1 = vpermute v1
(4) [r1 + r2] = str r3	r1 is a scalar; r2 is the induction variable		[r1 + r2] = vstr v3
(5) [r1 + r2] = str r3	r1 is a scalar; r2 is a vector; r2 has values loaded into it from an offset array		v3 = vpermute v3 [r1 + r2] = vstr v3
(6) r1 = dp r2, r3	r2 is a vector; r3 is a vector	r1 is a vector; size of r1 is recorded	v1 = vdp v2, v3
(7) r1 = dp r2, r3	r2 is a vector; r3 is a vector; r3 has values loaded into it	r1 is a vector; size of r1 is recorded	v1 = vdp v2, #const
(8) r1 = dp r2, r3	r2 is a vector; r3 is the induction variable (or vice-versa); r2 has values loaded into it	r1 is a vector; values loaded into r2 are copied to r1	None: this format is only used to update the induction variable for permutations.
(9) r1 = dp r1, r2	r1 is a scalar; r2 is a vector	r1 is a scalar	r1 = vred v2
(10) r1 = add r1, #1	r1 is the induction variable		r1 = add r1, SIMD_width
(11) any other instruction	all source operands are scalar		The input instruction is passed unmodified

Table 6.3: Rules used to dynamically translate the scalar code to SIMD code. dp refers to any data processing opcode, and vred refers to a vector opcode that reduces a vector to one scalar result (e.g., min).

occurring. Those previous values (i.e., the offset vector) are used to index a content addressable memory (CAM), and if there is a hit, the appropriate shuffle is inserted into the SIMD instruction stream. If the CAM misses, then the offset being loaded is a shuffle not supported in the SIMD accelerator and translation is aborted. Note that storing the entire 32 bits of previous values is unnecessary, because the values are only used to determine valid constants, masks, and permutation offsets; numbers that are too big to represent simply abort the translation process. The process of reading a source register's previous values, and conditionally writing them to the destination register, accounts for 11 of the 16 gates on the critical path.

Opcode Generation Logic: Once register state for an instruction's source operands has been accessed, it is passed to the opcode generation logic. Opcode generation logic uses simple combinational logic to determine how to modify an opcode based on the operands.

This essentially performs the reverse of the mapping described in Section 6.3, using rules defined in Table 6.3. For example, if the incoming instruction is a scalar load, then the opcode logic will write a vector load into the microcode buffer and tell the register state to mark the destination as a vector. Likewise, if the incoming instruction is an add, and the register state says both source registers are vectors, opcode generation logic will write a vector add into the microcode buffer and mark the destination register as a vector. A small amount of state is kept alongside this logic to recognize idioms of scalar instructions. Whenever an idiom is detected, this logic has the ability to invalidate previously generated instructions in the microcode buffer.

Opcode generation logic is fairly simple provided the SIMD instruction format is similar to the equivalent scalar instructions, since the scalar instructions require little modification before insertion into the microcode buffer. This is the case with our implementation, and thus the logic only takes up approximately 9000 cells. Control generation is not on the critical path in the current implementation, but it is very close to being critical. It likely would be on the critical path if there was not good correlation between baseline and SIMD instruction formats.

Microcode Buffer: The final component of the dynamic translator is the microcode buffer. This is primarily just a register array used to store the SIMD instructions until a region of scalar code has completed mapping. The maximum length of a microcode sequence was limited to 64 instructions in this implementation. Section 6.5 shows that this is sufficient for the benchmarks examined. At 32 bits per instruction, the microcode buffer contains 256 bytes of memory, which accounts for a little more than half of its 77,000 cells of die area. The rest of the area is consumed by an alignment network for collapsing instructions when idioms or permutations invalidate previously generated instructions.

Recall that the register state is used to detect when memory operations are indexed using a previously loaded offsets from constant arrays (Categories (7) and (8) in Table 6.1). When this situation is detected, the opcode generation logic will insert the appropriate permutation and memory instructions. At this point, the previously generated vector load of the offset vector can safely be removed. Removing this instruction while inserting multiple other instructions requires an alignment network. It should be noted that removing the

	Scalar Instruction	SIMD Generated
1	mov r0, #0	mov r0, #0
2	ld r1, [bfly + r0]	v1 = vld [bfly + r0]
3	add r1, r0, r1	
4	ld f0, [RealOut + r1]	vf0 = vfld [RealOut + r0] vf0 = vbfly vf0
5	ld f1, [ImagOut + r1]	vf1 = vfld [ImagOut + r0] vf1 = vbfly vf1
6	ld f2, [ar + r0]	vf2 = vfld [ar + r0]
7	ld f3, [ai + r0]	vf3 = vfld [ai + r0]
8	mult f2, f2, f0	vf2 = vmult vf2, vf0
9	mult f3, f3, f1	vf3 = vmult vf3, vf1
10	sub f6, f2, f3	vf6 = vsub vf2, vf3
11	ld f5, [RealOut + r0]	vf5 = vld [RealOut + r0]
12	sub f3, f5, f6	vf3 = vsub vf5, vf6
13	add f4, f5, f6	vf4 = vadd vf5, vf6
14	ld r2, [mask + r0]	v2 = vld [mask + r0]
15	and f3, f3, r2	vf3 = vmask vf3, #const
16	and f4, f4, r2	vf4 = vmask vf4, #const
17	ld r3, [bfly + r0]	v3 = vld [bfly + r0]
18	add r3, r0, r3	
19	str [tmp0 + r3], f3	vf3 = vbfly vf3 [tmp0 + r0] = vstr vf3
20	str [tmp1 + r0], f4	vf4 = vbfly vf4 [tmp1 + r0] = vstr vf4
21	add r0, r0, #1	r0 = add r0, #8
22	cmp r0, #128	cmp r0, #128
23	blt Top_of_loop_1	blt Top_of_loop_1

Table 6.4: Example translating scalar representation from Figure 6.4(B) back into SIMD instructions.

offset load is not strictly necessary for correctness, and eliminating this functionality would greatly simplify the microcode buffer.

After the microcode buffer receives the End signal from the legality checker, SIMD instructions are written into the microcode cache. SIMD code will then be inserted into the pipeline upon subsequent executions of the outlined function.

6.4.2 Dynamic Translation Example

To better illustrate how the dynamic translation hardware functions, Table 6.4 shows an example, translating the scalar loop in Figure 6.4(B) back into SIMD instructions for an 8-wide SIMD accelerator. The second loop from Figure 6.4(B) would be translated in a similar manner, and not refused with the original fissioned loop. Translation is very straight-forward for the vast majority of opcodes in the example, making the design of a hardware dynamic translator simple.

Instruction 1, the move, is the first instruction to enter the dynamic translator. As per the rules in Table 6.3, `r0` is marked as the induction variable in the register state, and the instruction is inserted into the microcode buffer unmodified.

Next, instruction 2 is translated. This is a load based on a scalar (the address `bfly`) and the induction variable (`r0`). Table 6.3 shows this is translated into a standard vector load. `R1` is marked as a vector and the value loaded is stored as a previous value of `r1` in the register state. After that, instruction 3 is translated. The register state shows that `r0` is the induction variable and `r1` is a vector with previous values associated with it. This instruction generates no instruction.

Now instruction 4 needs to be translated. Since one of the sources, `r1`, has previous values associated with it, this load may correspond to a shuffle instruction. The register state will look at the previous values, use them to CAM into a ROM and see that these offsets correspond to a known permutation instruction. In parallel, the load is being turned into a vector load by the opcode generation logic. Both of these instructions are inserted into the microcode buffer. Additionally, a pointer from the register state is used to remove the vector load created for instruction 2; a load of the offset is not necessary once the butterfly is inserted. This process of creating a load and shuffle is repeated for instruction 5.

Translating the remaining instructions in this example is just a matter of applying the rules presented in Table 6.3. Any instruction that does not match the rules defined in that table does not meet the proposed scalar virtualization format, and causes translation to abort. Once all scalar instructions have been translated, the outlined function returns, and

the microcode buffer writes the SIMD instructions into the microcode cache. This enables the SIMD code to be inserted into the instruction stream upon subsequent encounters of the outlined function.

6.5 Evaluation

To evaluate the Liquid SIMD system, an experimental framework was built using the Trimaran research compiler [121] and the SimpleScalar ARM simulator [9]. Trimaran was retargeted for the ARM instruction set, and was used to compile scalar ARM assembly code. The ARM assembly code was then hand-modified to include SIMD optimizations and conversion to the proposed scalar representation using a maximum targeted SIMD width of 16. Automatic SIMDization would have been used had it been implemented in our compiler. Again, automatic SIMDization is an orthogonal issue to abstracting SIMD instruction sets.

In our evaluation, SimpleScalar was configured to model an ARM-926EJ-S [5], which is an in-order, five stage pipelined processor with 16K, 64-way associative instruction and data caches. A parameterized SIMD accelerator, executing the Neon ISA, was added to the ARM-926EJ-S SimpleScalar model to evaluate the performance of SIMD accelerators for various vector widths. Simulations assumed dynamic translation took one cycle per scalar instruction in an outlined function. However, we demonstrate that dynamic translation could have taken tens of cycles per scalar instruction without affecting performance.

Liquid SIMD was evaluated using fifteen benchmarks from SPECfp2000 (171.swim, 179.art, 172.mgrid), SPECfp95 (101.tomcatv, 104.hydro2d), SPECfp92 (052.alvinn, 056.ear, 093.nasa7), MediaBench (GSM Decode and Encode, MPEG2 Decode and Encode), and common signal processing kernels (FFT, LU, FIR). The set of benchmarks evaluated was limited by applicability for SIMD optimization and the current capability of the ARM port of our compiler. None of these limitations were a result of the Liquid SIMD technique.

Dynamic Translation Requirements: In order to further understand the costs of Liquid SIMD, we first studied characteristics of benchmarks that impact design of a dynamic translator. One such characteristic is the required size of the microcode cache. The mi-

Benchmark	Mean	Max
052.alvinn	12.5	13
056.ear	34.5	36
093.nasa7	45.5	59
101.tomcatv	35.5	61
104.hydro2d	27.2	40
171.swim	37.8	51
172.mgrid	46.2	62
179.art	12.8	19
MPEG2 Dec.	12.5	13
MPEG2 Enc.	14.5	19
GSM Dec.	25	25
GSM Enc.	19.5	28
LU	11	11
FIR	11	11
FFT	31.3	38

Table 6.5: Number of scalar instructions in outlined function(s).

crocode cache is used to store the SIMD instructions after an outlined procedure call has been translated. This characteristic is also important for software-based translators, as it affects the size of code cache needed for the application.

We found that supporting eight or more SIMD code sequences (i.e., hot loops) in the control cache is sufficient to capture the working set in all of the benchmarks investigated. One question remaining then is how many instructions are required for each of these loops. With a larger control cache entry size, larger loops may be translated, ultimately providing better application performance. The downside is increased area, energy consumption, and latency of the translator. However, large loops that do not fit into a single control cache entry may be broken up into a series of smaller loops, which do fit into control cache. The downside of breaking loops is that there will be increased procedure call overhead in the scalarized representation. This section later demonstrates that procedure call overhead is negligible when using an 8-entry control cache.

Table 6.5 presents the average and maximum number of instructions per hot loop in the benchmarks. In some benchmarks, like 172.mgrid and 101.tomcatv, hot loops in the Trimaran-generated assembly code consisted of more than 64 instructions, and were broken into two or more loops. This decreased the number of instructions in each loop dramatically

Benchmark	< 150	< 300	> 300	Mean
052.alvinn	0	0	2	19984
056.ear	0	0	3	96488
093.nasa7	0	0	12	23876
101.tomcatv	0	0	6	16036
104.hydro2d	0	0	18	24346
171.swim	0	0	9	33258
172.mgrid	0	0	13	5218
179.art	0	0	5	2102224
MPEG2 Dec.	0	1	1	269
MPEG2 Enc.	0	3	1	257
GSM Dec.	0	0	1	358
GSM Enc.	0	0	1	538
LU	0	0	1	15054
FIR	0	0	1	13343
FFT	0	0	3	7716

Table 6.6: Number of cycles between the first two consecutive calls to outlined hot loops. The first three columns show the number of outlined hot loops that have distance of less than 150, less than 300, and greater than 300 cycles between their first two consecutive calls.

because it also reduced the number of load and store instructions caused due to register spills. Table 6.5 shows that 172.mgrid and 101.tomcatv have the largest outlined functions with a maximum of nearly 64 instructions. In most of these benchmarks, it would be possible to decrease the number of instructions per loop to less than 32 in order to decrease the size of the microcode cache.

These results lead us to propose a control cache with 8 entries of 64 SIMD instructions each. Assuming each instruction is 32 bits, this would total a 2 KB SRAM used for storing translated instruction sequences.

Another benchmark characteristic that affects dynamic translator design is latency between two executions of hot loops. Translation begins generating SIMD instructions for outlined scalar code the first time that a code segment is executed. If translation takes a long time, then SIMD instructions might not be available for many subsequent executions of that hot loop. This restricts the performance improvement achievable from a Liquid SIMD system. Moreover, if translation takes a long time, then the dynamic translator will need some mechanism to translate multiple loops at the same time.

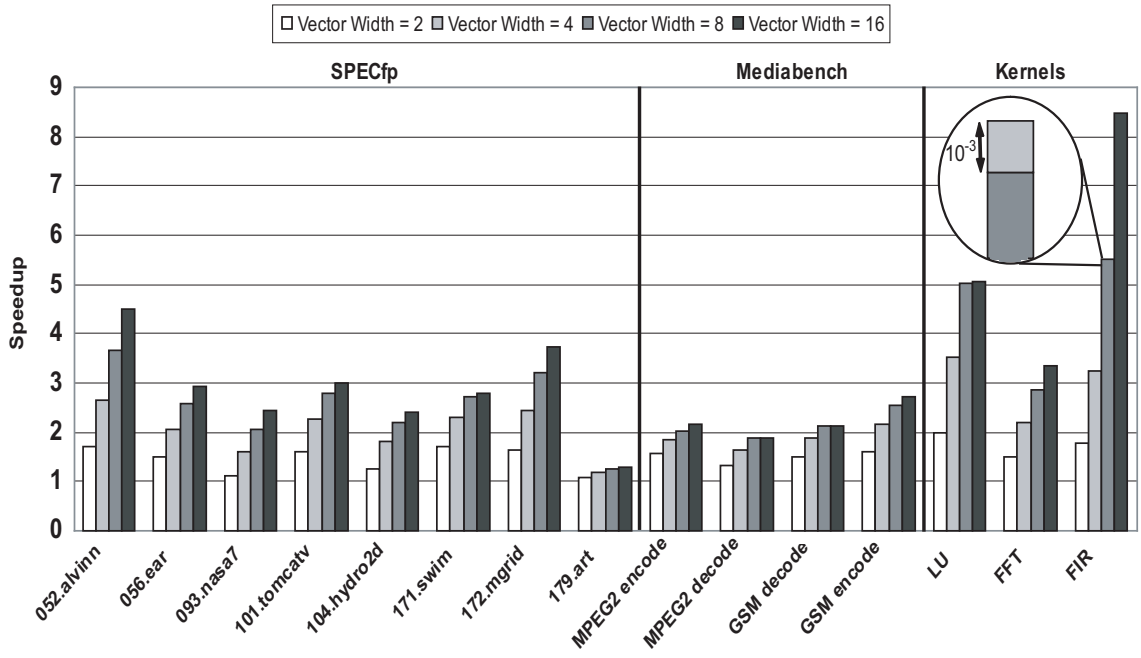


Figure 6.6: Speedup for different vector widths relative to a processor without SIMD acceleration. The callout shows the speedup improvement for a processor with built-in ISA support for SIMD instructions.

Table 6.6 shows the number of cycles between the two first consecutive calls to outlined hot loops for the benchmarks. In all benchmarks except MPEG2 Encode and Decode, there is more than 300 cycles distance between outlined procedure calls. The reason for large distances is that the scalar loops usually iterate several times over dozens of instructions, and also because memory accesses tend to produce cold cache misses. Table 6.6 shows that there is significant time for hardware based dynamic translation to operate without adversely affecting performance. A carefully designed JIT translator would likely be able to meet this 300 cycle target, as well.

Performance Overhead from Translation: Figure 6.6 illustrates the speedup attained using one Liquid SIMD binary (per benchmark) on machines supporting different width SIMD accelerators. Speedup reported is relative to the same benchmark running on a ARM-926EJ-S processor without a SIMD accelerator and without outlining hot loops. Compiling with outlined functions would have added a small overhead (less than 1%) to the baseline results.

In the ideal case, a SIMD-enabled processor with unlimited resources can achieve a speedup of $\frac{1}{\frac{S}{W}+(1-S)}$, where S is SIMD optimizable fraction of the code and W is the accelerator vector width. Some of the factors that decrease the amount of speedup in real situations are cache miss penalties, branch miss predictions, and trip count of the hot loop.

As expected, speedup generally increases by increasing the vector width supported in the SIMD hardware. In some of the benchmarks, like MPEG2 Decode, there is virtually no performance gain by increasing the vector width from 8 to 16. This is because the hot loop(s) in these benchmarks operate on vectors that are only 8 elements. Supporting larger vector widths is not beneficial for these applications. 179.art shows the least speedup of any of the benchmarks run. In this case, speedup is limited because 179.art has many cache misses in its hot loops. FIR showed the highest speedup of any benchmark because approximately 94% of its runtime is taken by the hot loop, the loop is fully vectorizable, and there are very few cache misses.

Figure 6.6 shows that SIMD acceleration is very effective for certain benchmarks. However, this fact has been well established and is not the purpose of this chapter. The main purpose of Figure 6.6 is to demonstrate the performance overhead of using dynamic translation in a Liquid SIMD system. Overhead stems from executing SIMD loops in their scalar representation whenever the SIMD version does not reside in the microcode cache. To evaluate the overhead, the simulator was modified to eliminate control generation. That is, whenever an outlined function was encountered, the simulator treated it like native SIMD code.

The performance improvement from using native instructions was measured for all fifteen benchmarks. Of these benchmarks, the largest performance difference occurred in FIR, illustrated in the callout of Figure 6.6. Native SIMD code provided 0.001 speedup above the Liquid SIMD binary. This demonstrates that the performance overhead from virtualization is negligible.

Code Size Overhead: Compilation for Liquid SIMD does increase the code size of applications. Code size overhead comes from additional branch-and-link and return instructions used in function outlining, converting SIMD instructions to scalar idioms, and also from aligning memory references to a maximum vectorizable length (discussed in

Section 6.3). Obviously, too much code size expansion will be problematic, creating instruction cache misses, which may affect performance.

To evaluate code size overhead, the binary sizes of unmodified benchmarks were compared with Liquid SIMD versions. The maximum difference observed occurred in hydro2d, and was less than 1%. The reason behind this is that the amount of SIMD code in the benchmarks is very small compared to the overall program size. Code size overhead is essentially negligible in Liquid SIMD.

6.6 Related Work

Many different types of accelerators have been proposed to make computation faster and more efficient in microprocessors. Typically, these accelerators are utilized by changing the instruction set; that is, statically placing accelerator control in the application binary. This means that the binary will not run on systems without that accelerator, or even systems where the accelerator has changed slightly.

To allow more flexibility in the instruction set, some previous work [28, 30, 60, 94, 112, 130] has recognized the benefits of dynamically binding instructions to an accelerator. Many different methods have been proposed to generate microcode for the various targeted accelerators at runtime. For example, work by Hu [59, 60] demonstrated the effectiveness of using binary translation software to dynamically generate control for one type of accelerator, a 3-1 ALU. The rest of these techniques utilize trace cache based hardware structures, to perform translation. Our method evolves this approach for SIMD accelerators.

There is a great deal more related work if the scope of dynamic binding is expanded to include benefits other than accelerator utilization. Dynamic binding has long been used to support modern microarchitectures in the context of legacy ISAs, such as the use of micro-ops (including micro-op fusing) in Intel processors [47]. Another motivation for dynamic binding has been to enable runtime optimizations. Several standard compiler optimizations, such as dead code elimination and constant propagation, benefit from runtime information available to dynamic translators [51].

Continuous Optimization [41] and RENO [100] are both examples of dynamic transla-

tors that perform traditional compiler optimizations by translating instructions during the decode stage of pipelines. The rePLay [99] project similarly optimized code, but operated on instructions post-retirement. Post-retirement translation is attractive because there is usually a long latency between instruction retirement and its next use [45], effectively taking translation off the critical path.

Just in time compilers (JITs) and virtual machines, such as Dynamo [10], DAISY [39], and the Transmeta Code Morpher [38], are all examples software-only dynamic translators. Software dynamic translators have been proposed both for code optimizations and to translate one ISA to another.

Virtualizing a SIMD ISA is similar to the way modern graphics related shader applications [19] are executed. In these applications, pixel and vertex shaders are distributed in an assembly-like virtual language such as DirectX, which has support for SIMD. At runtime, the shaders rely on a virtual machine to translate the virtual SIMD instructions into architecture-specific SIMD instructions. The benefits of using scalar instructions to virtualize SIMD instructions, as opposed a virtual language, is that a translator is not necessary to run the application.

The hardware translator proposed in this chapter is closely related to two other works [97, 123]. These papers developed methods to utilize SIMD hardware dynamically, without software support for identifying the instructions. That is, these works (often speculatively) create SIMD instructions from an arbitrary scalar binary. The hardware support required to perform this translation is generally more complicated than our proposed design, which merely recognizes and translates a set of predetermined instruction patterns.

The proposed hardware translator is also similar to work by Brooks [21] and Loh [84]. These papers propose using dynamic translation to detect when operations do not use the entire data path (e.g., only 8-bits of a 32-bit ALU), and then pack multiple narrow operations onto a single function unit.

Somewhat related to this chapter are the decades of research that have gone into automated compiler-based SIMDization. Many of these techniques are summarized by Krall [69] for the UltraSparc VIS instruction set, and by Bik [14] for Intel's SSE instructions. Recent work [40, 127] has investigated techniques to vectorize misaligned memory references

through data reorganization in registers. Other recent work [73] introduced techniques to extract vector operations within basic blocks and selective vectorization of instructions. Automatic SIMDization is completely orthogonal to the work in this chapter; the SIMD virtualization scheme proposed here can be used in conjunction with or in the absence of any automated SIMD techniques.

The main contribution of this chapter is the development of a method for virtualizing SIMD instructions in a way amenable to dynamic translation. No previous work has done this. To demonstrate that our virtualization schema is easily translated, the design of a post-retirement hardware translator was presented in Section 6.4. Any other style of dynamic translator could have been used to prove this point, though.

6.7 Summary

Liquid SIMD is a combination of compiler support and dynamic translation used to decouple the instruction set of a processor from the implementation of a SIMD accelerator. SIMD instructions are identified and expressed in a virtualized SIMD schema using the scalar instruction set of a processor. A light-weight dynamic translation engine binds these scalar instructions for execution on an arbitrary SIMD accelerator during program execution. This eliminates the problems of binary compatibility and software migration that are inherent to instruction set modification.

This chapter presented a software schema powerful enough to virtualize nearly all SIMD instructions in the ARM Neon ISA using the scalar ARM instruction set. The design of a hardware dynamic translator was presented, proving that the software schema is translatable and that this translation can be incorporated into modern processor pipelines. Synthesis results show that the design has a critical path length of 16 gates and the area is less than 0.2 mm^2 in a 90 nm process. Experiments showed that Liquid SIMD caused code size overhead of less than 1%, and performance overhead of less that 0.001% in the worst case. This data clearly demonstrates that Liquid SIMD is both practical and effective at solving the compatibility and migration issues associated with supporting multiple SIMD accelerators in a modern instruction set.

CHAPTER 7

Design and Utilization of Cyclic Accelerators

7.1 Introduction

Previous chapters in this dissertation describe the design and utilization of acyclic and SIMD accelerators. The purpose of this chapter is to extend those ideas to accelerators targeting computation in the form of innermost loop bodies. The benefit of accelerating entire loop bodies, instead of just acyclic portions, is that more work is done in hardware, making the resultant execution more efficient. The downside of doing more work in hardware is that the accelerator is less programmable; that is, fewer applications are able to take advantage of the accelerator because the class of computation accelerated is more specialized.

Accelerators targeting innermost loops present a good design point in the efficiency versus programmability spectrum. Many applications spend the majority of their time executing in innermost loops, meaning that accelerators targeting this type of computation can potentially be broadly applicable. Additionally, there are several characteristics of innermost loops (discussed in Section 7.2.1) that make hardware implementations particularly efficient. The accelerators described in this chapter are more efficient, but less programmable, than the acyclic accelerators discussed in previous chapters.

The first part of this chapter presents the architectural exploration and design of a hardware accelerator that targets a class of loop bodies for a wide range of applications. By defining a single architecture to accelerate loops, the recurring costs of designing an application-specific accelerator are eliminated. The goal is to cost-effectively generalize

an ASIC design to make it useful for a wider range of loops (i.e., increase the programmability), without generalizing it to the point where it begins to look like a general purpose processor.

The second step is to attack the software costs of targeting a cyclic accelerator. As with acyclic and SIMD accelerators, software costs result from re-engineering the application once the underlying hardware has changed. To avoid these costs we develop a software abstraction that virtualizes the salient architectural features of loop accelerators. An application that uses this abstraction is dynamically retargeted to take advantage of the accelerator if it is available in the system; however, the application will still execute correctly without any accelerator in the system. The tradeoff is to abstract away as many architecture-specific features as possible without requiring a significant overhead to dynamically retarget the application.

There are two primary contributions of this chapter:

- It presents the design a novel loop accelerator architecture. Design space exploration ensures that the accelerator design is broad enough to accelerate many different applications, yet very efficient at executing the targeted style of computation.

- It describes an dynamic algorithm for mapping loops onto loop accelerators. The algorithm is analyzed to determine the runtime overheads introduced by this dynamically mapping loops, and static/dynamic tradeoffs are investigated to mitigate the overhead.

7.2 Overview

It is widely acknowledged that the vast majority of execution time for most applications is spent in loops. Applying this fact, along with Ahmdal's Law, generally leads system designers to construct hardware implementing loop bodies whenever ASICs are needed to meet performance or power consumption goals. For example, special purpose loop accelerators for Fast Fourier Transforms and Viterbi decoding are ubiquitous in modern embedded SoCs.

This section begins by describing the general architecture common across loop accelerators. Next, it gives an overview of *modulo scheduling*, a compilation technique used to schedule loops so that they effectively use the hardware resources available to them. The section concludes by introducing the issues surrounding dynamically retargeting applications to a particular loop accelerator implementation, which are discussed in detail in Section 7.4.

7.2.1 Loop Accelerator Architectures

In order to determine an appropriate architecture for a broad set of loop bodies, it is first necessary to identify the general structure of loop accelerators. Figure 7.1 shows the high level structure of a loop accelerator. At the top of this figure, address generators stream data into the accelerator. The address patterns typically follow a simple, deterministic pattern (often based on the loop's induction variable(s)) that enables them to be decoupled from the computation performed on the data. When data is streamed in from the memory system, it is placed in FIFOs that are accessed by function units (FUs). Address generators can be time multiplexed to fetch multiple streams, which enables them to hide any stalls due to bursty memory behavior amongst the different streams. Input data that is not streamed into the accelerator, such as constants or scalar inputs, are written into a register file. Typically, this register file is memory mapped and must be initialized before invoking the accelerator.

Once all the data is available, FUs begin processing it, reading values from the FIFOs or register file and writing results to either the output memory buffers or registers. The register file that stores results from the FUs, need not be a monolithic standard SRAM; many loop accelerators utilize distributed SRAMs [29] or more efficient structures such as FIFOs [43] or ShiftQs [1]. Additionally, the functionality provided by the FUs is often highly customized, executing several RISC equivalent operations back-to-back in the name of improved efficiency [116].

Once computation has completed, another set of address generators stream the results back to memory. It is assumed that the input and output memory streams are mutually exclusive, so that the accelerator does not need to perform memory dependence analysis.

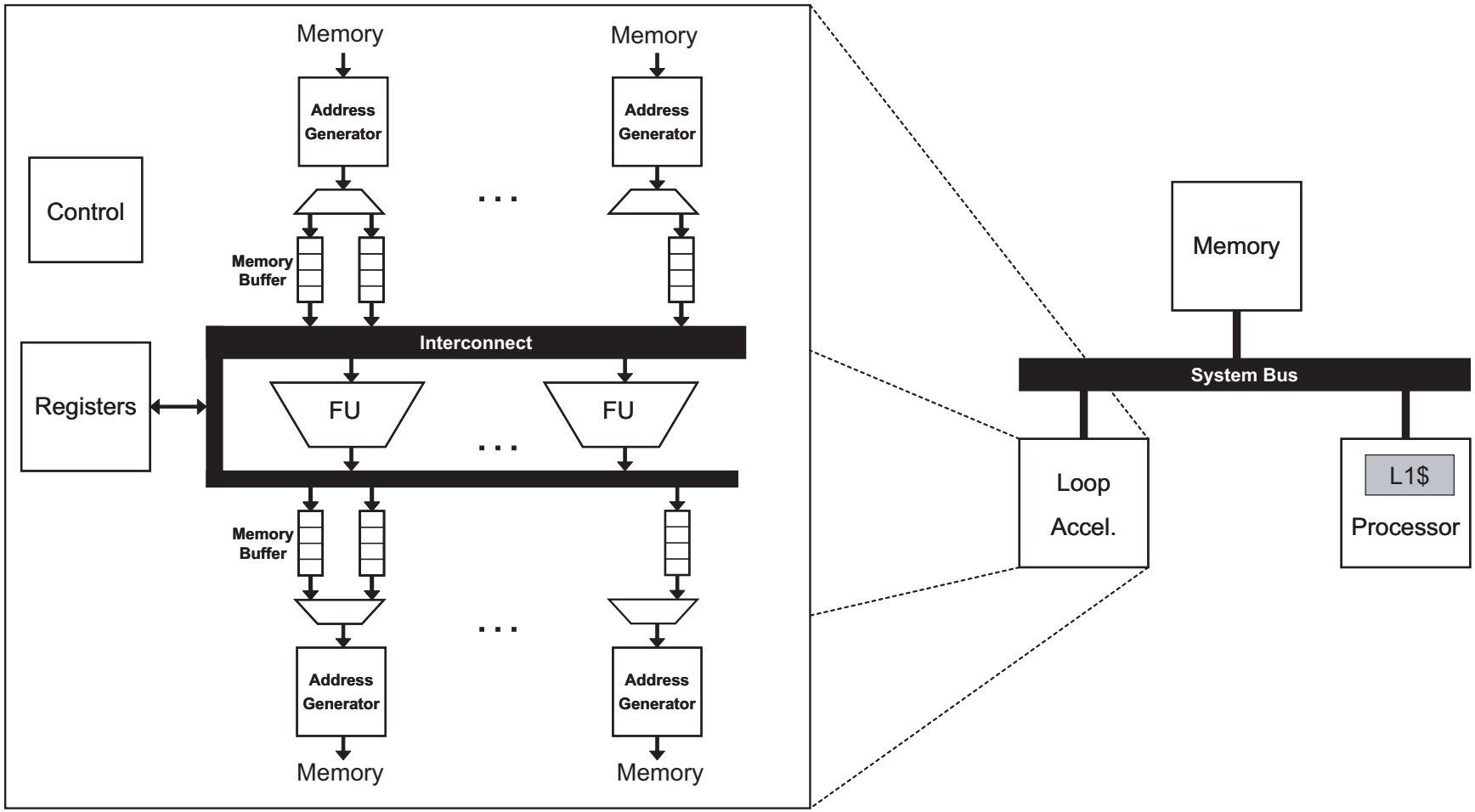


Figure 7.1: An architecture template for loop accelerators

This abstract architecture encapsulates the structure of most loop-targeting ASICs [114] as well as previously proposed generalized loop accelerators [26, 29, 88].

There are several reasons why this architecture is more efficient at executing loops than general purpose processors. First, the control flow in loops is very simple, removing the need for control flow speculation such as sophisticated branch predictors. Second, the repeating control sequence (instructions) used to configure the accelerator can be stored in a circular buffer, which is much more efficient to access than a large instruction cache. Third, the memory accesses are not data dependent and are implicitly independent from each other, enabling memory accesses to be decoupled from the computation and obviating the need for dependence analysis. Lastly, the interconnect, FUs, and register files can be customized to fit the needs of the application or domain that is being targeted.

7.2.2 Utilizing Loop Accelerators

Assuming that there is an effective piece of hardware for executing loops, it is also necessary to have a capable compilation strategy to make use of the hardware. Modulo scheduling is a state-of-the-art software pipelining heuristic for scheduling loops, and provides the basis for the software techniques presented in this chapter. Previous work on modulo scheduling is extensive [33, 72, 74, 82, 83, 105–108, 110], and the purpose of this section is only to introduce fundamental concepts.

Figure 7.2 shows a sample modulo schedule. In this example, each iteration of the loop has six instructions (represented by grayed boxes), and there are three different FUs that can execute the instructions. The instructions are assigned to FUs so that new iterations can begin executing at a constant rate, called the *initiation interval*, or simply *II*.

Walking through the example in Figure 7.2, iteration 1 begins executing at cycle 0, and a new iteration begins every 2 cycles (the II) until all the FUs become fully utilized in cycles 4 through 7. After cycle 7, there are no more iterations to begin and so the software pipeline begins to drain until execution completes. The periods where the software pipeline is ramping up and ramping down are called the *prologue* and *epilogue*, respectively, and the steady state (when an iteration is starting and completing every II cycles) is called the

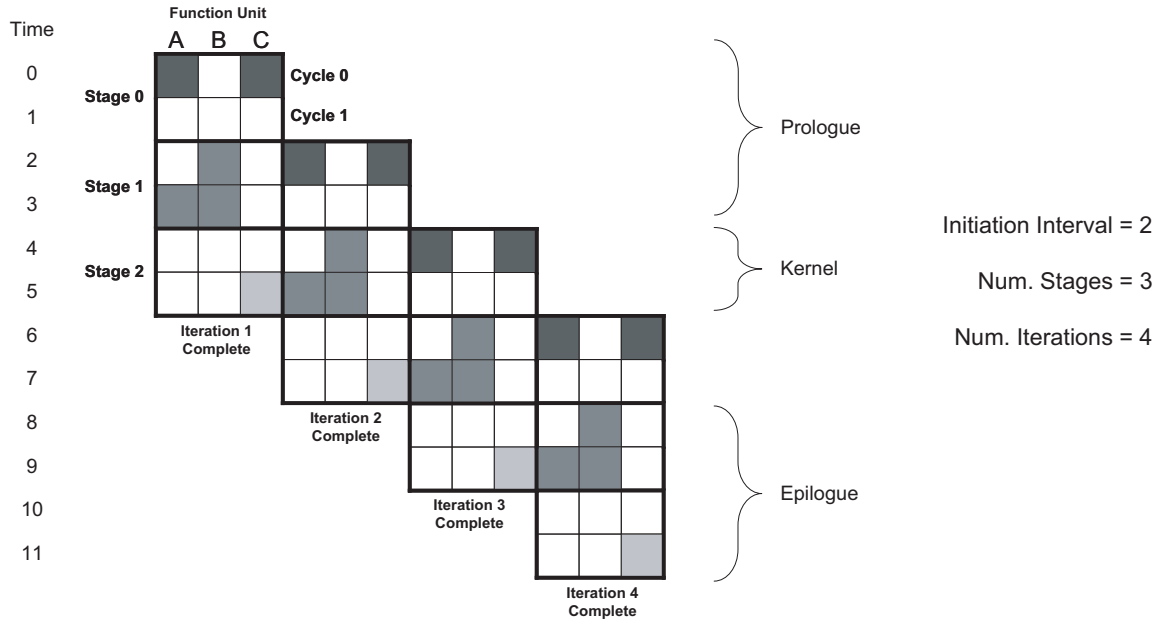


Figure 7.2: Important concepts in modulo scheduling loops

kernel.

A single loop iteration can be broken down into multiple *stages* based on how many times II cycles has passed since it began executing. For example, during cycle 6, iteration 2 is executing stage 3 instructions, iteration 3 is executing stage 2 instructions, and iteration 4 is executing stage 1 instructions. The different time steps in each stage are referred to as the stage *cycles*, which range from 0 to II-1. The goal of modulo scheduling heuristics is generally to make II as low as possible, so that kernel execution is reached and completed as soon as possible. A secondary goal is to make the number of stages (often abbreviated *SC* for stage count) as small as possible. To rephrase using standard pipeline terminology, lower II equates to higher iteration throughput and lower SC equates to lower latency.

Modulo scheduling has proven to be a very effective technique for software pipelining, however there are some limitations with the process. One limitation is that loops with function calls cannot be modulo scheduled. This problem can mitigated through intelligent function inlining, and is not a major drawback. A more important limitation is that while-loops and loops with side exits require special hardware support, such as speculative

memory accesses [91, 105]. Although it is feasible to support while-loops and loops with side exits, we chose to preclude them from this study, to minimize the architectural impact outside the accelerator itself.

Figure 7.3 demonstrates the implication of this decision. Each bar in this figure represents the entire execution time for a given benchmark from MediaBench or SPEC. The black bars on the bottom are the fraction of time spent executing in modulo schedulable loops. The bars labeled "Speculation Support" refer to the time spent in while-loops that would be modulo schedulable, provided the appropriate hardware support existed. Bars labeled "Subroutine" are loops with function calls that could not be inlined (e.g., calls into the math library that were not visible to the compiler).

Media processing and floating point applications (the left portion of Figure 7.3) tend to spend the vast majority of their execution time in modulo schedulable loops. Lack of support for loops requiring speculation will limit the utility of the loop accelerator for some applications (e.g., the applications on the right portion of Figure 7.3); however, modulo schedulable loops clearly represent an important class of computation worthy of hardware acceleration.

7.2.3 Dynamic Retargeting for Binary Compatibility

Using specialized hardware to execute loops has many performance and power benefits, but hardware and software design costs prevent widespread deployment in many cases. Designing one architecture for a broad set of loops tackles the hardware design costs. However, the software design costs remain a difficult problem.

Software costs arise from the fact that the control that invokes a loop accelerator is statically encoded in the binary. An application that utilizes an accelerator typically has no forward or backward compatibility. This means that whenever the underlying hardware platform changes, the application must completely re-engineered.

The method proposed to avoid these software costs is to virtualize the accelerator interface. That is, we will analyze the steps used during compilation to map applications onto loop accelerators, and perform as much of it dynamically as possible. This enables the

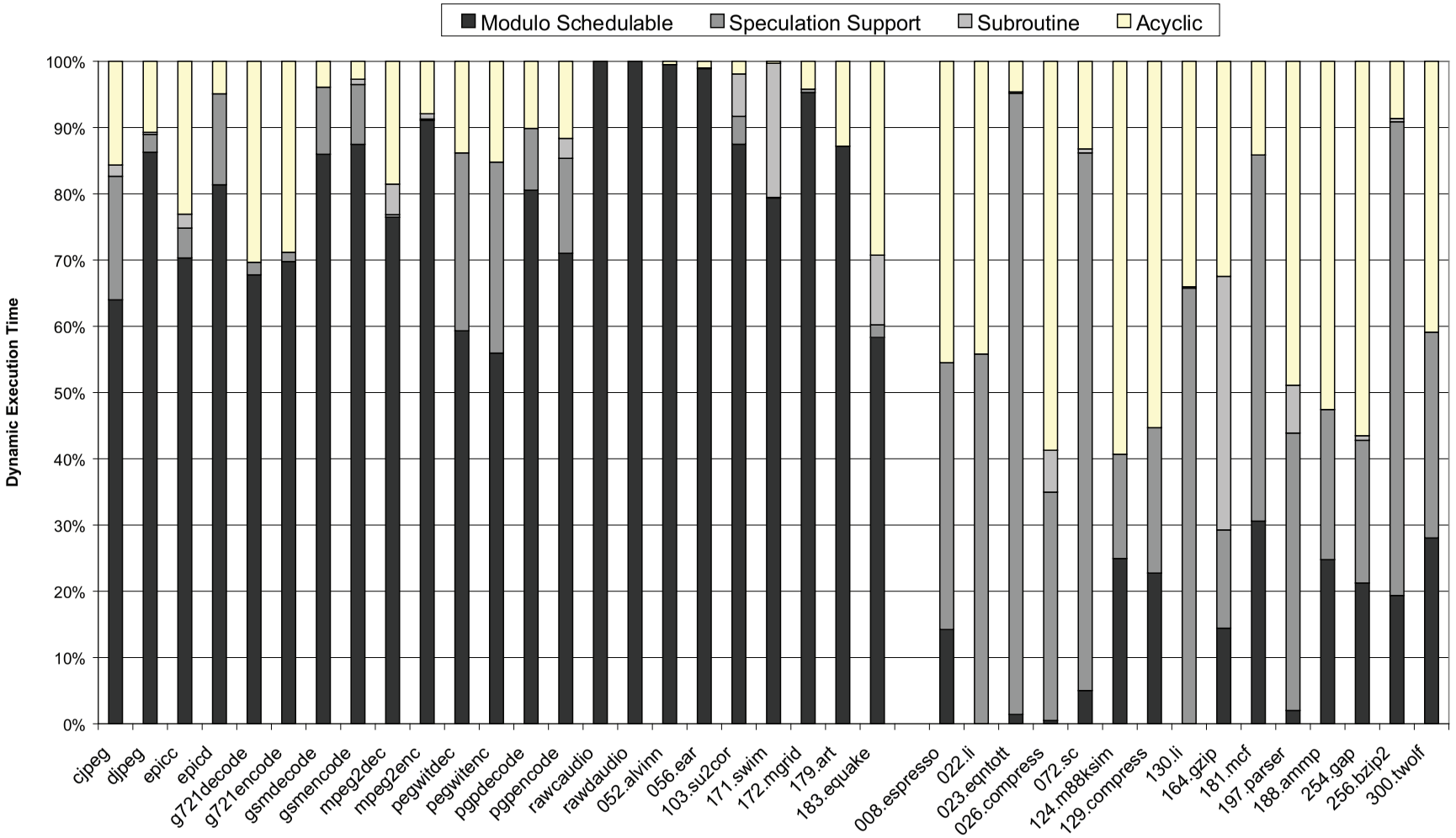


Figure 7.3: Percent of execution time spent in various types of code. “Speculation Support” refers to while-loops and loops with side exits, “Subroutine” refers to loops that have a non-inlinable function call, and “Acyclic” refers to code not known to be in a loop.

binary to be flexible, not tying it to any one specific accelerator architecture.

The challenge in virtualization is to determine the appropriate static/dynamic tradeoffs to make in the binary. High quality modulo scheduling heuristics can be sophisticated, taking too long to fully perform dynamically. If the translation takes too long, it can completely erode all the efficiency benefits from using the accelerator in the first place. At the other end of the spectrum, performing the mapping entirely statically ties the binary to a single accelerator implementation, which has significant non-recurring engineering costs if the underlying hardware changes.

The remainder of this chapter is organized as follows: Section 7.3 performs a design space exploration for a generalized loop accelerator for a range of media and floating point applications. This design provides the basis for our work on virtualization, which is covered in Section 7.4. Section 7.4 walks through the details of one particular modulo scheduling heuristic and analyzes the tradeoffs involved in performing each step statically versus dynamically.

7.3 Generalized Loop Accelerator

To mitigate the costs of customized hardware, it often makes sense to extend the programmability of ASICs, making them more useful across a broader set of applications. The goal of this section is to do just that: design an architecture that effectively supports the set of modulo schedulable loops from the MediaBench and SPECFP applications on the left portion of Figure 7.3. Designing this architecture has two purposes. First, we provide a quantitative analysis of the tradeoffs involved with adding each execution resource to the accelerator. Previous work [26, 88] designing generalized loop accelerators presented designs without this analysis. Second, this design helps us gauge the static/dynamic tradeoffs in modulo scheduling to target a loop accelerator. It has been reported that the time needed to modulo schedule a loop strongly correlates to the number of resources in the target machine [33], and so a representative architecture is necessary to accurately measure translation overheads.

The loop accelerator architecture template shown in Figure 7.1 will serve as the basis

for our generalized design. Customizing the template for the targeted application set now requires identifying how many resources of each type these applications require. To determine this, we modified the Trimaran toolset [121] to compile for and simulate a processor with attached loop accelerator. The accelerator connects to the processor through a system bus using a memory mapped interface.

7.3.1 Design Space Exploration

The baseline architecture in our design space exploration assumes a hypothetical loop accelerator with infinite resources. That is, we modulo schedule loops onto a machine with unlimited registers, execution units, memory ports, etc. Architectural parameters were then individually varied to determine what fraction of the infinite-resources speedup was attainable using finite resources. The Swing modulo scheduling heuristic [82] was used to target each application to the accelerators. More details about the Swing modulo scheduler are presented in section 7.4.1. As previously mentioned, only the benchmarks on the left portion of Figure 7.3 were used in this analysis.

Figures 7.4 and 7.5 show the results of the design space exploration for execution units and register requirements. The x-axis in these graphs represents the number of resources available in the system and the y-axis is the fraction of infinite-resource speedup attained. For example, the gray line in Figure 7.5 shows that when there is only one floating point register, the average speedup across the targeted application suite is 60% of what is attainable with infinite floating point registers.

Figure 7.4 explores the function units available in the accelerator, where IEx and FEx represent integer and floating point execution units, respectively. One interesting result from this experiment was that very few floating point units (FEx in the right graph) were needed to attain the a significant amount of speedup in the application set. This is partially due to the significant number of integer-only applications in our target suite, but the long latency of floating point operations also contributes to this result. If a floating point unit is fully pipelined (which was assumed) modulo scheduling does a very good job utilizing the unit every possible cycle.

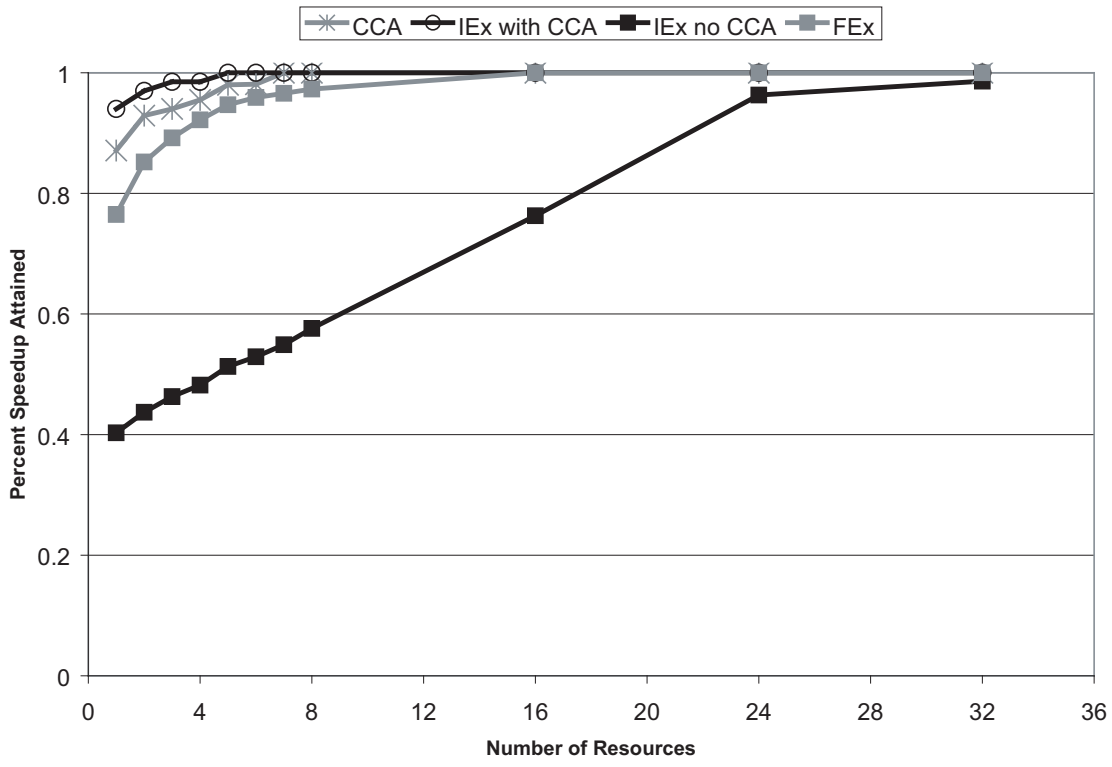


Figure 7.4: Execution resource needs. Each line is the fraction of infinite-resource loop accelerator speedup attained when varying the number of execution units

One surprising result from Figure 7.4 is that the point of diminishing returns for integer execution units is very high, on the order of 24 units. Due to this result, we chose to experiment with another type of function unit, a CCA [28]. The CCA (shown in Figure 7.6) is a logic structure specifically designed to efficiently implement the most common types of integer computations. It supports 4 inputs, 2 outputs, and can execute as many as 15 standard RISC operations atomically in 2 clock cycles. The primary benefits of the CCA result because it executes much larger pieces of computation as a group, reducing storage and interconnect requirements, as well as squeezing more work out of each clock cycle. The top line in Figure 7.4 shows that when one CCA is added to the loop accelerator, the required number of integer execution units drops dramatically.

Figure 7.5 shows the required number of registers needed to store live-ins, live-outs, constants, and temporary values for the loop. Overall, few registers are needed to support

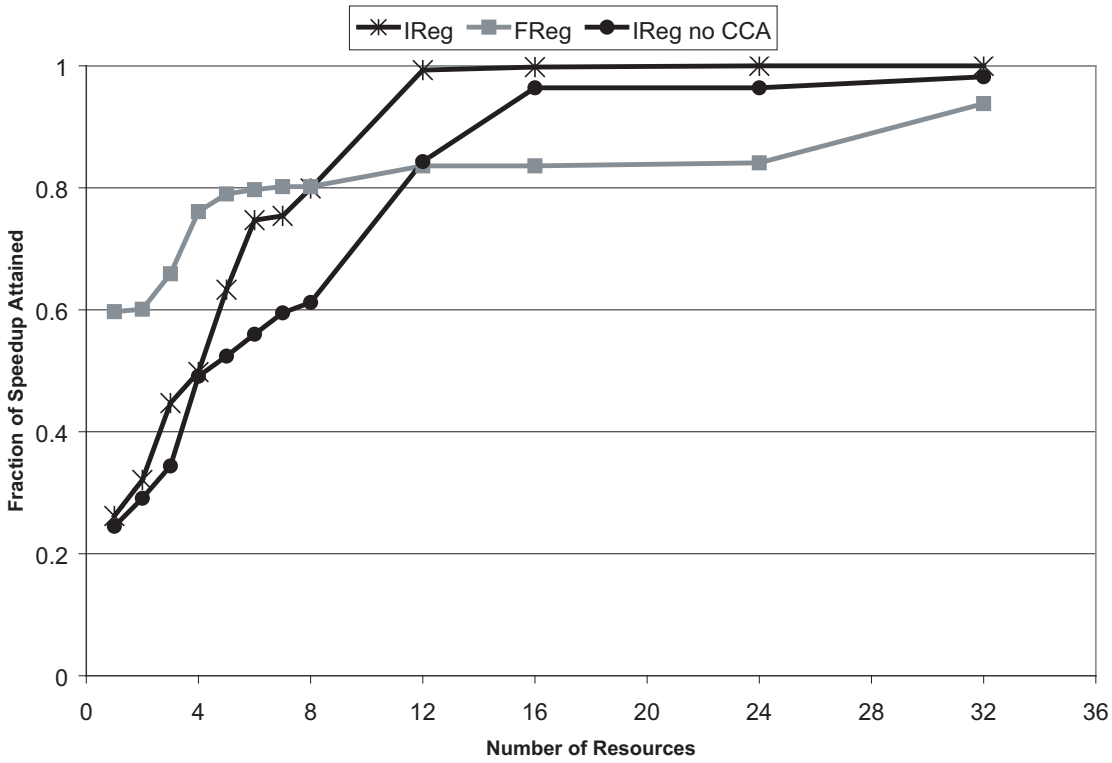


Figure 7.5: Register file resource needs. Each line is the fraction of infinite-resource loop accelerator speedup attained when varying the number of registers

the majority of important loops. As would be expected, adding a CCA to the system reduces the register requirements, since fewer temporaries are needed to communicate between separate execution units.

Similar to Figures 7.4 and 7.4, Figure 7.7 shows the the fraction of infinite-resource speedup attained when varying a particular resource. This graph varies the number of load/store streams supported in the accelerator. As would be expected, loads are more important than stores. Surprisingly, many loops can be supported without any address generators streaming data out to memory; these loops only have scalar outputs, which are read directly from the memory mapped register file upon loop completion.

Another surprising result from the memory stream analysis is that a very large number of memory streams were needed to support several important loops in the examined benchmarks. For comparison purposes, previously proposed general loop accelerators only

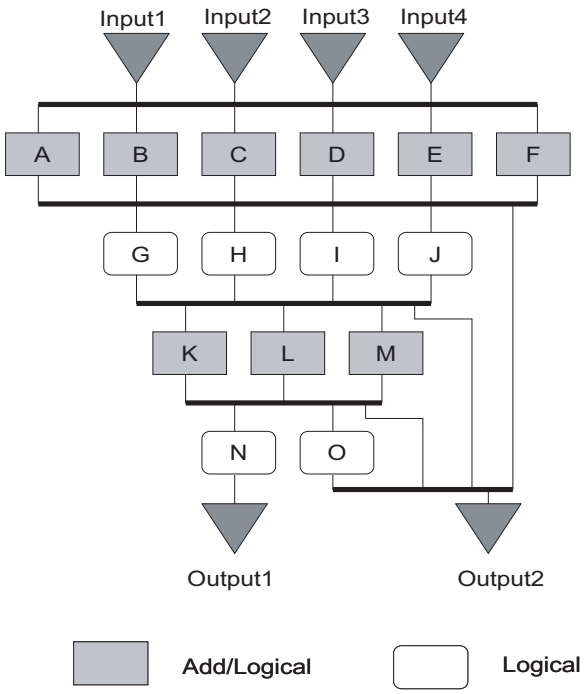


Figure 7.6: CCA execution unit from [28]

supported 3 load/1 store [26] stream or 6 total load/store streams [88] per loop. Supporting fewer memory streams is desirable, since it requires less hardware.

Empirically speaking, the loops that required a large number of memory streams tended to be very large. One potential way to reduce the hardware overhead of supporting these loops is to time-multiplex the address generators. Large loops tend to have larger IIs, giving the address generators time to process several different streams. Another potential solution is to break the large loops up into smaller loops using a technique such as decoupled software pipelining [96]. This would reduce the required number of streams for each individual loop but increase memory traffic, as dividing the loop up would likely create communication streams between the smaller loops.

The graph in Figure 7.8 shows the maximum supported II by the loop accelerator (i.e., loops that cannot be scheduled in at the maximum II will not be accelerated). This is an important consideration in the accelerator design, because the size of the loop control is directly proportional to the maximum supported II. Recall from Figure 7.2 that, in the steady-state, the kernel of the loop simply repeats over and over. Since the kernel is II cycles

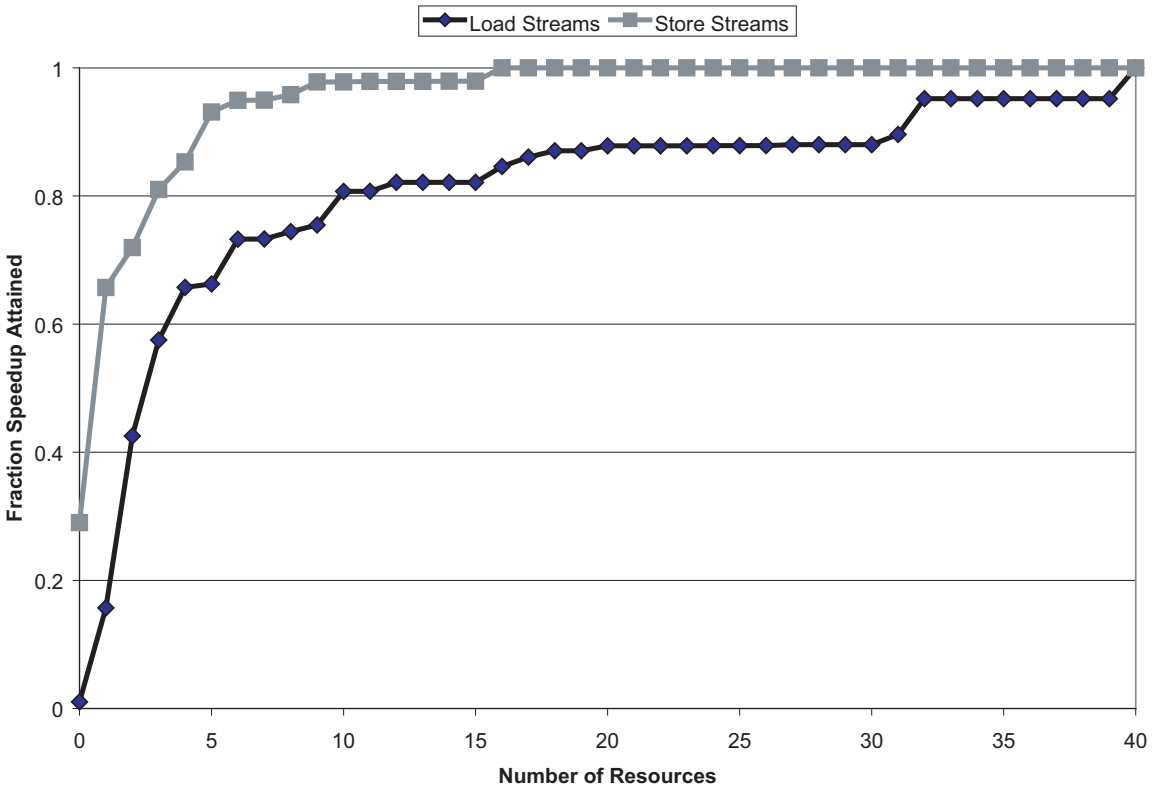


Figure 7.7: Memory stream resource requirements

long, II determines the size of the control structure, assuming there is support to selectively disable stages for the prologue and epilogue. As with the memory stream limitation, if a particular loop is too large to be supported by an II, often times using the compiler to split the loop into multiple smaller loops will enable the loop to utilize an accelerator.

Using the analysis in this section as a guide, we propose a generalized loop accelerator design consisting of 1 CCA, 2 integer units (including multipliers), 4 floating point units, 16 floating point and integer registers, 16 load memory streams (time-multiplexed among 4 address generators), 8 store memory streams (multiplexed among 2 address generators), and a maximum II of 16. This is sufficient for attaining 83% of the speedup possible using a hypothetical loop accelerator with infinite resources.

The design space exploration presented here has omitted two major portions of the data path: the register file structure and interconnect customizations that often occur in

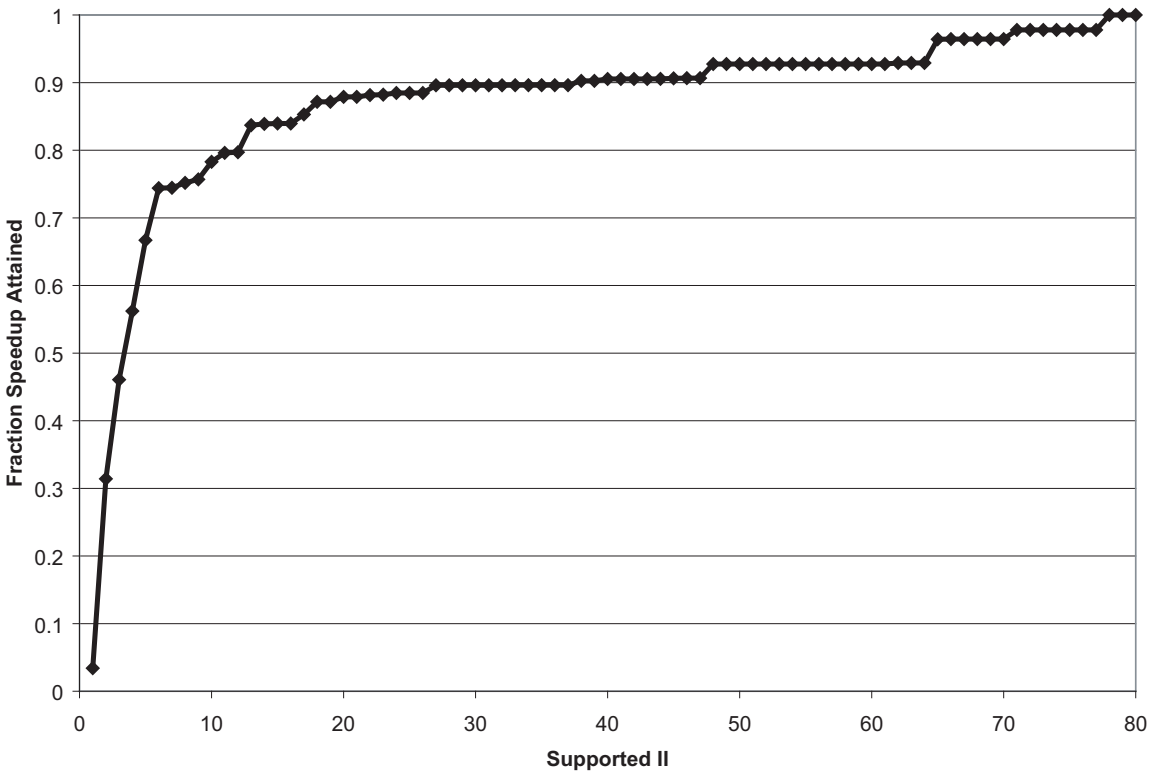


Figure 7.8: Impact of maximum supported II on potential speedup

customized hardware accelerators. The primary reason for this omission is that there are currently few modulo scheduling algorithms that take these customizations into consideration. Without software support to analyze the costs of architectural customization (in terms of reduced performance) it is difficult to make intelligent design decisions, and so we leave this exploration for future work.

7.3.2 Loop Accelerator Control

Figure 7.9 shows the control logic used to support modulo scheduled loops in the accelerator. This design is very similar to previous work [65, 91] that used this type of control for modulo scheduling support in digital signal processors (DSPs) and general-purpose VLIW processors. The bits needed to configure each part of the loop accelerator are stored in control stores shown in the top right of Figure 7.9. One of these control stores is necessary

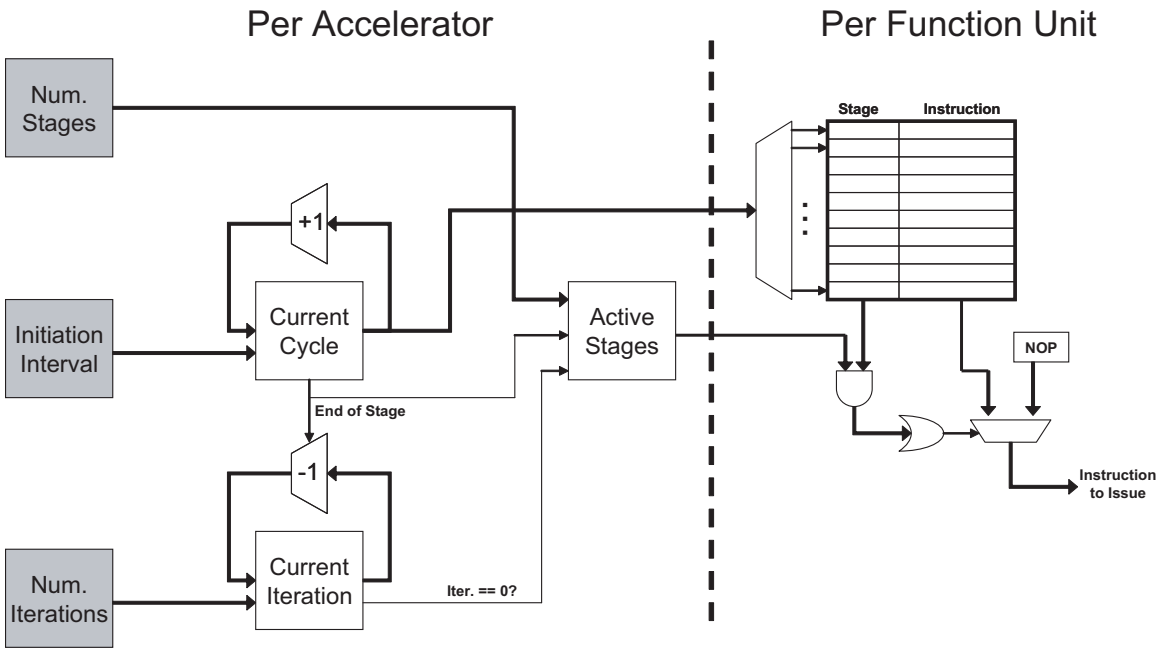


Figure 7.9: Control logic in the loop accelerator

for each architectural element (e.g., FU) that can be scheduled individually. Each row in the control store corresponds to a particular stage cycle. The instructions are tagged with each stage they belong to. For example, if this memory stored the control for function unit C in Figure 7.2, then the first entry would have a dark gray instruction tagged stage 0 and the second entry would have a light gray instruction tagged stage 2.

The gray boxes at the left of Figure 7.9 completely define which instructions execute at a given time. Every clock cycle, the Current Cycle is incremented. When the Current Cycle reaches II, 1 stage of the loop has completed, so Current Cycle is reset to 0, and the Current Iteration is decremented. In order to support prologue and epilogue execution, the Active Stages bit vector is used to disable instructions whose stage is not supposed to execute.

Using the loop in Figure 7.2 as an example, software configures the loop accelerator control by writing 3 into Num. Stages, 2 into the Initiation Interval, 4 into Num. Iterations. Instructions and a bit vector representing each instruction's stage are also written into the control stores on a per function unit basis. Figure 7.10 walks through the execution of FU C from Figure 7.2. Current Cycle starts at 0 and will access the first instruction from the

Time	Current Cycle	Current Iteration	Active Stages	Instruction
0	0	3	001	
1	1	3	001	
2	0	2	011	
3	1	2	011	
4	0	1	111	
5	1	1	111	
6	0	0	111	
7	1	0	111	
8	0	0	110	
9	1	0	110	
10	0	0	100	
11	1	0	100	

Stage	Instruction
001	
100	

Figure 7.10: Control logic walkthrough

control store (shown at the right in Figure 7.10). Active Stages is initialized to 001, since at the beginning of the loop only stage one is active. The Active Stages bit vector will be compared with the stage tag from the dark grey instruction accessed from the control store and the instruction will be issued, provided its stage bit is enabled. A similar process will happen when Current Cycle is 1 during the next clock cycle: the light grey instruction is accessed, the stage tag (100) is compared with the Active Stages bit vector, and the instruction is not issued, because its stage has not been activated. At the end of this cycle, Current Cycle is reset to 0, since the cycle iterates between 0 and II-1. This also causes the End of Stage signal to be sent to Current Iteration and Active Stages. Current Iteration is decremented, and a 1 is shifted into the Active Stages bit vector (011) enabling stage 1, because Current Iteration is non-zero. This process will continue until Current Iteration reaches zero, at which point zeros will start to be shifted into Active Stages (the epilogue). The loop has completed once Active Stages is all zeros.

7.4 Virtualizing the Loop Accelerator

The loop accelerator architecture is very effective at executing the modulo scheduled loops from the wide range of applications studied. However, the tradeoffs made in that design will not fit all situations. When this is the case, a new accelerator must be designed for the system, which creates a burden on the application developer. Traditionally, control used to invoke an accelerator is statically placed in the binary, meaning the application will have to be re-engineered to function on a different hardware platform. This software porting cost often prevents the deployment of specialized hardware in situations where it otherwise would provide benefits.

The way to eliminate the software cost is to generate the control for the accelerator dynamically, only after the application knows what accelerators are available in the system. Dynamic control generation relies on the assumption that the cost of performing the translation is low; otherwise the translation cost would outweigh any benefits provided by the custom hardware. Thus, the key to virtualization of custom hardware is analyzing the algorithms used to generate control, performing the time consuming parts statically, and encoding them in the binary in a way that is binary compatible with other systems.

Towards this end, this section will analyze the Swing modulo scheduling heuristic [82], with the goal of using this algorithm dynamically to map loops onto a loop accelerator.

From a system level view, we are proposing that the compiler statically mark modulo schedulable loops in applications. This can be accomplished either through a new instruction signaling the beginning of a loop or via a special encoding scheme, such as the procedural abstraction proposed in [31]. Once the loops are marked, a run time software system, such as Dynamo [10] or DAISY [39], will use the Swing algorithm to retarget the binary to utilize an available accelerator. The translated loops are stored in a software managed code cache, and the loop accelerator is used whenever possible. If a particular loop is not supported by the target accelerator (for example, if the II required was too high), then the translation simply aborts and the loop can execute on the general-purpose processor.

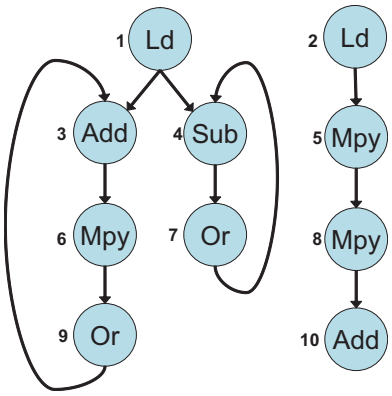


Figure 7.11: An example loop body. For illustration purposes, assume multiplies take 3 cycles and all other operations take 1 cycle.

7.4.1 Dynamically Mapping using Swing Modulo Scheduling

Swing modulo scheduling is a heuristic for software pipelining loops [82]. This algorithm is used as the basis for dynamic loop mapping because previous work [33] demonstrated that it produces high quality schedules and is significantly faster than other modulo scheduling algorithms, particularly when the machine has a large number of resources.

There are many steps in mapping a loop onto a loop accelerator. First, if a CCA is present in the system, the translator tries to collapse multiple RISC instructions into a single CCA instruction. The CCA is designed to efficiently execute larger pieces of integer computation, and so moving computation to this resource improves the loop schedule. Optimally utilizing the CCA is an NP-complete problem [57], so this work uses a greedy algorithm to keep runtime overheads low.

After CCA mapping, modulo scheduling begins. The first step is to compute the minimum II that the loop could potentially be scheduled at. The minimum II is a function of both the recurrences in the loop and the resources available in the accelerator. As an example, consider the loop in Figure 7.11. This loop has two recurrences, ops 3-6-9 which is 5 cycles long and ops 4-7 which is 2 cycles long. Since the longest recurrence is 5 cycles long, II must be at least 5, since it is impossible to start future iterations before the recurrence completes execution. Resources may also affect the minimum II for the loop in Figure 7.11. For example, assume the target loop accelerator had only one multiplier. Since

there are 3 multiply instructions in the loop, II must be at least 3 because an iterations worth of computation must be issued every II cycles. The minimum II for a loop is the maximum of the recurrence and resource constrained IIs. A more thorough discussion of algorithms to compute II is covered in [105].

Now that II is computed, Swing prioritizes operations to determine the order in which to schedule them. Simplifying a bit, the Swing priority function tries to schedule the most critical recurrence first, moving through less critical recurrences, and then finally to operations that do not appear on a recurrence path. The intuition behind this is that scheduling the recurrences is a more constrained problem since operations have a min and max schedule time. Additionally, failing to schedule a recurrence at a given II will make the schedule fail, forcing the scheduler to increase II (lowering performance) in order to map the loop. Using Figure 7.11 as an example again, the Swing priority will try to schedule the most critical recurrence, 3-6-9, followed by the next most critical recurrence, 4-7, followed by the remaining acyclic operations. Once the operations are prioritized, Swing uses a slightly modified list scheduling algorithm to assign each operation to a slot in the modulo schedule. Full details of the Swing prioritization and scheduling algorithms are found in [82].

After a loop schedule is generated, a postpass maps operands from the virtual loop encoding to the register files/memory buffers in the loop accelerator. If there are not enough registers to support the translated loop, translation aborts, and the loop is executed on the baseline processor. In addition to operand mapping, the translator must also generate instructions to move scalar inputs/outputs between the loop accelerator and the scalar processor.

7.4.2 Evaluation

In order to gauge the overheads associated with dynamically mapping loops onto an accelerator, the Swing algorithm was implemented as a post-pass to compilation in the Trimaran toolset. The number of instructions needed to retarget each loop was recorded using OProfile [78], which reads on-chip performance counters; the average penalty per loop is reported in Figure 7.12.

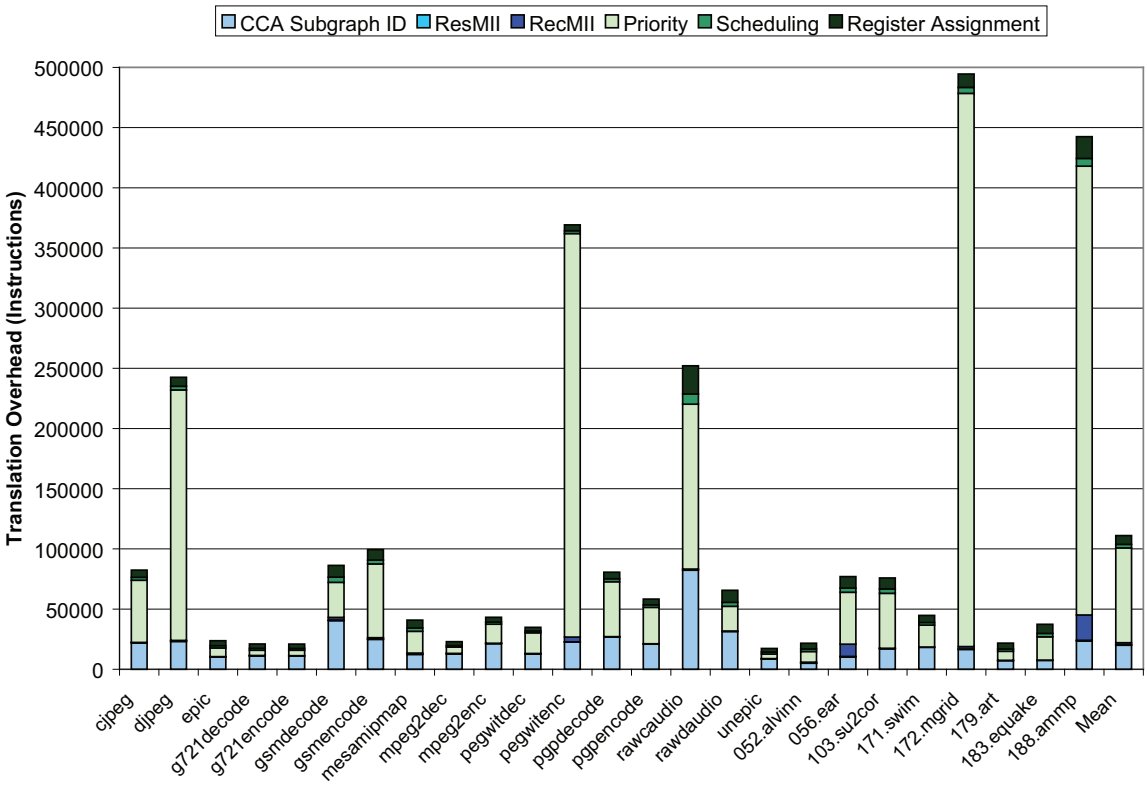


Figure 7.12: The measured translation overhead per loop.

There are a few important trends to take away from this graph. First, the average loop translation time varies widely from benchmark to benchmark. The primary reason for this is that the number and size of the loops also varies by a large factor, and larger loops require more work to translate. A secondary reason for the high variance is that the algorithm used in Swing's priority calculation takes significantly more time if there are many recurrences in the loop. Applications that took the longest time to translate did not necessarily have the largest loops.

The most important take-away from Figure 7.12 is the distribution of time spent in various phases of the Swing algorithm. On average, it took approximately 110,000 instructions to map each loop onto the targeted loop accelerator. 70% of those instructions were devoted to calculating the priority used in scheduling, and 18% of the instructions were spent mapping subgraphs onto the CCA. The vast majority of translation time was spent performing

these two tasks, which motivates us to perform these steps statically.

One potential, non-static solution to reducing the priority run time is to use a simpler priority function. A promising candidate is to use the height-based priority function proposed in [105]. The simpler priority function was previously found to be effective in [105], because of the more exhaustive backtracking scheduler used in the algorithm. However, using the simpler priority function in conjunction with the single-pass scheduler in Swing often yielded sub-optimal schedules.

Statically encoding the Swing priority for each operation in the binary is another solution. One of the fortunate characteristics of the Swing priority function is that it focuses on scheduling the most critical recurrences in the loop first, and recurrences are architecture independent[1]. Statically encoding priority in the binary enables a high quality schedule, while at the same time reducing the average loop translation time from 110,000 down to 36,000 instructions. One potential way to encode this is by creating a data section in the binary immediately before the loop. One number is encoded for each instruction in the loop, and once the size of the loop is known the priorities are easily recoverable. Statically encoding the instructions that map onto the CCA has been covered in previous work [28], and further reduces the translation overhead from 36,000 down to an average of 17,000 instructions per loop.

Figure 7.13 demonstrates the importance of driving the translation overhead as low as possible. This graph shows the average speedup attained when varying the translation cost per loop. The various lines reflect how frequently the translation penalty must be paid. For example, the top line assumes that each modulo schedulable loop need only be translated once during benchmark execution, and the bottom line assumes each loop must be translated 10% of the time when it is invoked, due to eviction from a code cache.

If translation costs average 110,000 instructions per loop (as shown in Figure 7.12) and each instruction takes one cycle, even a 1% miss rate severely impacts the speedup provided from the loop accelerator: moving from an idealized speedup of 2.63 down to 1.47. Driving the translation penalty down to 17,000 instructions, by performing the CCA

[1] It should be noted that the criticality of recurrences are only architecture independent if execution latencies of the function units remain consistent across the architectures.

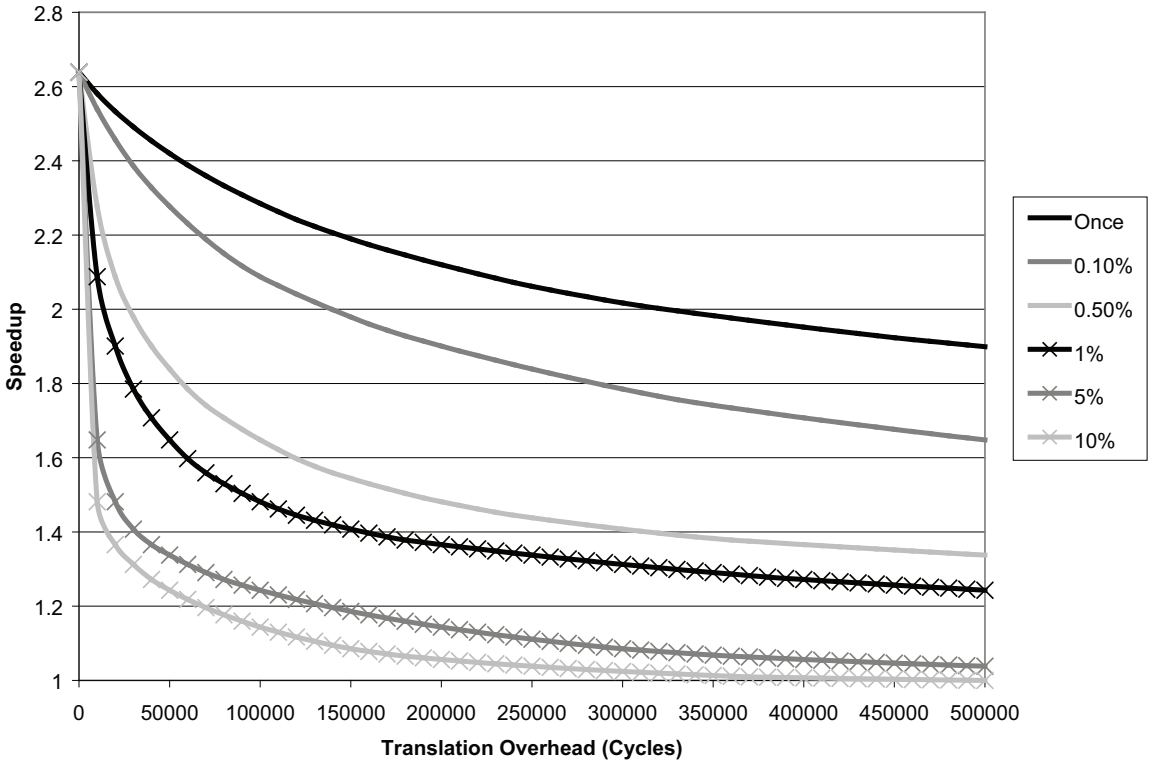


Figure 7.13: Speedup attained when varying the translation overhead penalty. Each line represents how frequently the penalty must be paid.

selection and priority computation offline, would push the average speedup up to almost 2. An alternate way to view Figure 7.13, is that it stresses the importance of providing enough space in the code cache so that loops do not need to be repeatedly translated.

Figure 7.14 shows the speedup over a single-issue processor measured per application using a realistic code cache. It was assumed the code cache provided enough space to store the previous 16 translated loops using an LRU eviction policy. Using the target architecture proposed in Section 7.3, this works out to approximately 48 KB of storage, which is small compared with typical code cache sizes [55]. The hit rates for each application varied slightly, but all were very close to 100%.

The left-most bar for each application shows the speedup from using the loop accelerator, assuming no translation penalty. This is equivalent to the speedup of a statically compiled binary. The next bar, labeled “Swing Real Cache”, shows the speedup when as-

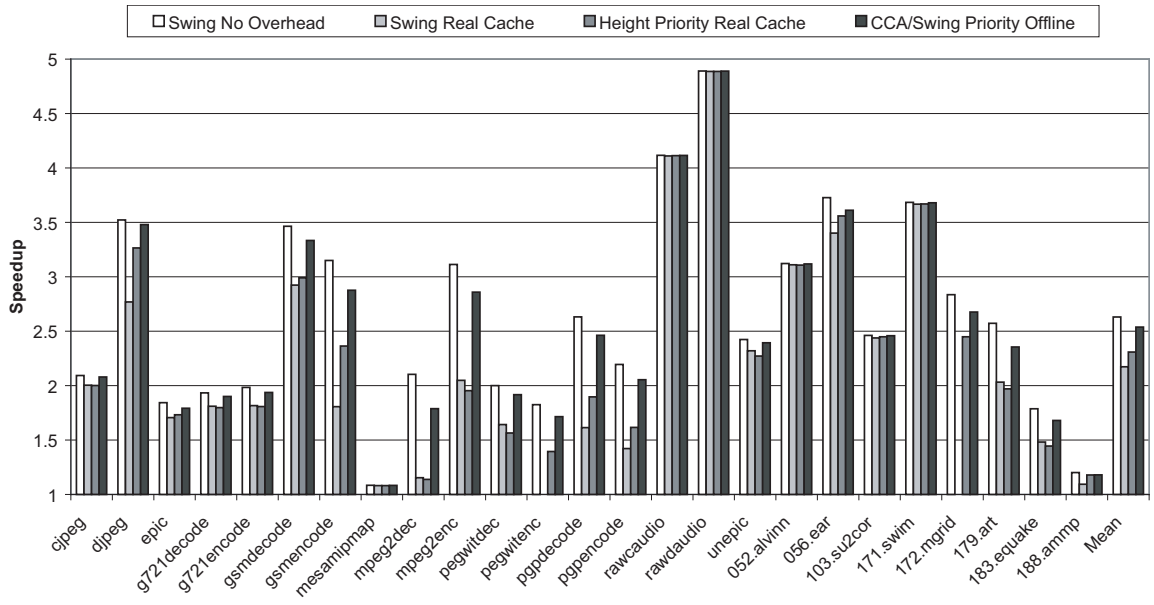


Figure 7.14: Static/dynamic and algorithm tradeoffs for the key mapping stages.

suming a realistic translation cache and the penalties measured from preforming the entire Swing algorithm dynamically. The “Height Priority” bar is also fully dynamic, but instead uses the simpler height-based priority function. The final bar represents the speedups when CCA mapping and Swing priority calculation are performed offline and encoded in the binary.

Several interesting patterns emerge from Figure 7.14. First, for many benchmarks, such as rawcaudio, the translation overhead of performing Swing entirely dynamically has a negligible impact on the loop accelerator’s speedup (comparing the first two bars). In the case of rawcaudio, there is only one critical loop in the application and so the translation cost is easily amortized. Other applications showed little performance degradation because their most critical loops were quite small, making the translation costs negligible. The translation overhead for many other loops was quite significant, however. Mpeg2dec notably went from a speedup of 2.1 down to 1.15, and pegwitenc and 172.mgrid lost all performance benefits from the loop accelerator. On average, factoring the translation costs brought the speedup from 2.63 down to 2.17.

The middle two bars for each application show the tradeoff involved in using the Swing priority function in comparison with the simpler height-based priority. The less sophisticated height-based priority function sometimes generates schedules with higher IIs (and thus, worse performance), but the translation times are significantly faster. On average, the benefits of faster translation time outweighed the benefits of better schedules, providing a speedup of 2.3 compared with 2.17.

The final bar in Figure 7.14 shows that by moving the particularly difficult portions of mapping loops offline, the speedups can approach that of natively compiled code. On average, performing CCA mapping and Swing priority calculation offline reduced translation penalties to the point where the average speedup was 2.54 as compared with 2.63 for natively compiled code. This hybrid static/dynamic mapping strategy provides a significant 24% and 37% more speedup up over fully dynamic solutions utilizing height-based and Swing priority functions, respectively.

7.5 Related Work

As mentioned in previous chapters, accelerators are a popular method to increase the performance and efficiency of microprocessor designs. Several people have proposed accelerators specifically targeting loop nests, because the regular control structure in loops provides significant efficiency gains over processors designed for general purpose control structures. The Reconfigurable Streaming Vector Processor (RSVP) [26] is a vector-based accelerator designed for loops in multimedia applications running in an embedded environment. The architecture is similar to what we have proposed; however, RSVP uses SIMD execution units, and a single SRAM to buffer memory accesses. Mathew et al. propose another loop accelerator architecture in [88], which is very similar to the architecture proposed here. The main difference is the memory buffering structure and type of execution resources provided. This chapter goes beyond these two previous works by providing a quantitative analysis of accelerator resource needs using loops from a diverse application set. Other work, such as [91, 107], proposed adding hardware to a standard pipeline to efficiently support the control structure of loops. The control in our proposed accelerator is

very similar to [91], but our work extends this by additionally customizing execution and memory resources. The loop accelerator architecture presented in this chapter was primarily developed to provide a realistic target for evaluating dynamic mapping algorithms.

Statically generating efficient code for loops is also an area of much related work. Software pipelining [72] has proven to be an excellent way to improve the resource utilization of loop execution. Lam [72] showed that developing an optimal software pipelining is an NP-complete problem, and so many heuristics have been developed to produce high-quality schedules in a reasonable amount of time [33, 50, 74, 82, 83, 105, 106, 110, 118]. Most pertinent related work is the Swing Modulo Scheduling algorithm, originally proposed in [82]. Later work [33] demonstrated that this algorithm produces high quality schedules in much shorter runtimes than other modulo scheduling algorithms, making it a good starting point for dynamically retargeting loops. While the work in this chapter did not exploit this fact, Swing has been extended to support loops with complex control flow, such as side exits [74]. The contribution of this chapter is evaluating Swing Modulo Scheduling in the context of dynamically targeting a loop accelerator. The relative runtime of each modulo scheduling stage is measured, and we explore the tradeoffs associated with statically encoding the results of each stage in the binary.

Abstracting the underlying hardware structure to improve efficiency without affecting binary compatibility has much related work, as well. Perhaps the best known example of this is the Transmeta Code Morphing Software [38], which dynamically converts x86 application into VLIW programs. Dynamo [10], Daisy [39], and DIF [94] are all examples that dynamically translate applications to target entirely different microarchitectures. Several proposals exist to only translate select portions of an application to target accelerators. For example, [30, 59, 112] all explored the benefits of dynamically binding applications to acyclic accelerators. Other work [31] looked at dynamically binding for SIMD accelerators. This chapter is the first proposal for dynamically binding to cyclic accelerators.

7.6 Summary

Adding customized hardware to a processor is an effective way to improve the performance and efficiency of the system. However, significant hardware and software non-recurring engineering costs prevent customized hardware from being adopted in many situations. This chapter addresses those costs in the context of cyclic computation. Cyclic computation accelerators are a compelling design point, because they encompass a larger fraction of many applications' execution time than acyclic accelerators, even though cyclic accelerators are not as broadly applicable as acyclic ones.

This chapter presented the design of a generalized loop accelerator. Design space exploration was used to ensure that the accelerator is applicable to a wide range of media and floating point applications. This generalized design provides a good architecture for executing common modulo schedulable loops, thus eliminating the engineering costs associated with designing loop-specific accelerators from scratch.

Software costs were addressed by virtualizing the accelerator interface. Modulo scheduled loops are statically marked in the binary and expressed in the baseline instruction set. At runtime, a dynamic translator attempts to map the loop onto any available accelerators using modulo scheduling. This work found dynamically modulo scheduling loops has a significant performance overhead and proposed statically encoding scheduling priority to be an effective technique for minimizing the overhead. Overall, the loop accelerator and dynamic compilation system provided a mean speedup of 2.54 over a single-issue processor, and the resulting binary remains flexible enough to be used by systems with different (or even no) accelerators.

CHAPTER 8

Summary

Industry has produced, and consumers rely on, continual exponential performance increases from microprocessor-based systems. The traditional method for improving performance, increasing clock frequency, is no longer effective as sharply rising power consumption has made designs with higher clock frequencies too expensive to cool. This trend has given rise to a new generation of designs with many simpler cores on a single chip. Multicore chips attempt to improve performance by providing increased parallelism to applications. This strategy is well suited for some application domains, such as transaction processing; however, many applications without readily apparent parallelism suffer from this design decision.

The focus of this dissertation is on an alternate method to improve performance: hardware customization. Hardware customization is an attractive alternative to homogeneous multicore chips, because customized hardware is far more efficient than general-purpose hardware, and often applicable when coarse-grained parallelism is difficult or impossible to find. Technology trends, such as increasing transistor density and alternate manufacturing techniques [66], only serve to make hardware customization more likely in future designs. The aim of this dissertation was to solve many of the architectural and compilation challenges that traditionally made customized hardware difficult to implement.

One problem with customized hardware is the effort needed to design an accelerator for each target application or domain. Chapter 2 solves this problem for acyclic accelerators, by developing an automated technique for identifying the critical computation patterns

given an application (or set of applications). Hardware is synthesized specifically for the targeted applications without user input. The system demonstrated significant speedups for several applications, with as much as 2.39 and an average of 1.69, while utilizing modest additional die area.

Another contribution from Chapter 2 is the observation that critical computation subgraphs within a domain tend to be very similar, although they do not exactly match. This trend can be exploited, by slightly generalizing the functionality in the accelerators proactively. For relatively little additional cost, generalizing accelerators provides a substantial likelihood of the accelerators being useful even in the context of future algorithms.

Designing execution resources for one application or domain of applications makes sense in many high volume-markets, but non-recurring engineering costs render this technique infeasible in many other markets. To increase computational efficiency in these situations, Chapter 3 uses critical subgraph identification to design two general-purpose acyclic accelerators, one based on combinational logic and one based on lookup-tables. A wide range of applications are analyzed to ensure the most common computation patterns are efficiently supported in the proposed designs. Overall, average speedups of 1.66 for the combinational logic and 1.47 for the lookup-table based accelerators were achieved.

Another challenge in utilizing custom hardware is the engineering costs associated with integrating accelerators into existing hardware and software systems. The root of the problem stems from the typical method software uses to invoke accelerators, by statically encoding specialized control sequences into the application binary. This is costly because adding new accelerators implies that the hardware must understand the new control sequences. Software must also be re-engineered to include these control sequences, and if the underlying accelerator ever changes, the software will no longer be compatible.

Chapters 4 and 6 address this problem in the context of acyclic and SIMD accelerators, respectively. We apply the technique of delayed binding, where computation to be accelerated is expressed using the baseline instruction set of the processor. At runtime, a dynamic translator converts these sequences into accelerator-specific control sequences. This technique eliminates the need for new control sequences to be added to the hardware and software systems when adding customized hardware. To prove the utility of

dynamic binding, we present the design of several dynamic translators and evaluate the static/dynamic tradeoffs of performing some translation steps offline. Overall, we found that using dynamic binding enables accelerator integration in a binary compatible manner, while incurring negligible overheads in terms of die area, code size, and application slow down.

Chapter 7 builds on the work in previous chapters by discussing the design and integration of cyclic accelerators into microprocessor-based systems. Designing cyclic accelerators is a fundamentally harder problem because it requires handling memory references, residual state, long latency communication and many other aspects that were not issues in dealing with acyclic and SIMD accelerators. Additionally, dynamically binding an application onto a cyclic accelerator is algorithmically much more difficult. The analysis in Chapter 7 culminates in a generalized cyclic accelerator design and dynamic mapping algorithm, which is binary compatible, and provides 2.53 speedup in the average case.

A last problem related to customized hardware is how to automatically compile an application to target a particular accelerator. That is, given an application written in a high-level language, how do we pick out the portions that would be more effectively run on an accelerator? Chapter 5 solves this problem in the context of acyclic accelerators. Automatically generating code to optimally target accelerators is NP-hard, and the typical industry solution involves either hand coding or simplistic greedy algorithms. This dissertation presented a graph-based algorithm, which on average provided 50% more speedup than greedy solutions, while retaining the fast runtimes associated with greedy solutions.

As a whole, this dissertation has developed many solutions that enable customized hardware to be the vehicle for exponential performance growth needed in future processor designs.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Shail Aditya and Michael Schlansker. ShiftQ: A buffered interconnect for custom loop accelerators. In *Proc. of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 158–167, November 2001.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[3] A.V. Aho, M. Ganapathi, and S.W.K. Tijang. Code generation using tree pattern matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[4] A. Alomary et al. PEAS-I: A hardware/software co-design system for ASIPs. In *European Design Automation Conference*, pages 2–7, 1993.

[5] ARM Ltd. *ARM926EJ-S Technical Reference Manual*, January 2004. http://www.arm.com/pdfs/DDI0198D_926_TRM.pdf.

[6] M. Arnold. *Instruction Set Extensions for Embedded Processors*. PhD thesis, Delft University of Technology, 2001.

[7] Kubilay Atasu, Laura Pozzi, and Paolo Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proc. of the 40th Design Automation Conference*, pages 256–261, June 2003.

[8] P. M. Athanas and H. S. Silverman. Processor reconfiguration through instruction set metamorphosis. *IEEE Computer*, 26(3):11–18, 1993.

[9] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Transactions on Computers*, 35(2):59–67, February 2002.

[10] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proc. of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, June 2000.

[11] M. Baleani et al. HW/SW partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In *Proc. of the Tenth International Conference on Hardware/Software Codesign*, pages 61–66, May 2002.

[12] Max Baron. Cortex-A8: High speed, low power. *Microprocessor Report*, 11(14):1–6, 2005.

[13] J. P. Bennett. *A Methodology for Automated Design of Computer Instruction Sets*. PhD thesis, University of Cambridge, 1988.

[14] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the intel architecture. *International Journal of Parallel Programming*, 30(2):65–98, 2002.

[15] Partha Biswas, Sudarshan Banerjee, Nikil Dutt, Laura Pozzi, and Paolo Ienne. ISEGEN: Generation of high-quality instruction set extensions by iterative improvement. In *Proc. of the 2005 Design, Automation and Test in Europe*, pages 1246–1251, 2005.

[16] P. Bose and E. S. Davidson. Design of instruction set architctures for support of high-level languages. In *Proc. of the 11th Annual International Symposium on Computer Architecture*, pages 198–206, June 1984.

[17] A. Bracy, P. Prahlad, and A. Roth. Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 18–29, December 2004.

[18] R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, C-31(3):260–264, 1982.

[19] Mauricio Breternitz, Herbert Hum, and Sanjeev Kumar. Compilation, Architectural Support, and Evaluation of SIMD Graphics Pipeline Programs on a General-Purpose CPU. In *Proc. of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 135–145, 2003.

[20] P. Brisk et al. Instruction generation and regularity extraction for reconfigurable processors. In *Proc. of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 262–269, 2002.

[21] David Brooks and Margaret Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *Proc. of the 5th International Symposium on High-Performance Computer Architecture*, page 13, 1999.

[22] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[23] Jorge E. Carrillo and Paul Chow. The effect of reconfigurable units in superscalar processors. In *Proc. of the 9th ACM Symposium on Field Programmable Gate Arrays*, pages 141–150, 2001.

[24] H. Choi et al. Synthesis of application specific instructions for embedded DSP software. *IEEE Transactions on Computers*, 48(6):603–614, June 1999.

[25] Yuan Chou, Pazhani Pillai, Herman Schmit, and John Paul Shen. Piperench implementation of the instruction path coprocessor. In *Proc. of the 33rd Annual International Symposium on Microarchitecture*, pages 147–158, December 2000.

[26] Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi. The reconfigurable streaming vector processor (RSVP). In *Proc. of the 36th Annual International Symposium on Microarchitecture*, pages 141–150, 2003.

[27] N. Clark, H. Zhong, and S. Mahlke. Automated custom instruction generation for domain-specific processor acceleration. *IEEE Transactions on Computers*, 54(10):1258–1270, 2005.

[28] Nathan Clark et al. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 30–40, December 2004.

[29] Nathan Clark et al. OptimoDE: Programmable accelerator engines through retargetable customization, August 2004. In *Proc. of Hot Chips 16*.

[30] Nathan Clark et al. An architecture framework for transparent instruction set customization in embedded processors. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 272–283, June 2005.

[31] Nathan Clark et al. Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping. In *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, pages 216–227, 2007.

[32] Nathan Clark, Hongtao Zhong, and Scott Mahlke. Processor acceleration through automated instruction set customization. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, pages 129–140, December 2003.

[33] J. Codina, J. Llosa, and A. Gonzalez. A comparative study of modulo scheduling techniques. In *Proc. of the 2002 International Conference on Supercomputing*, pages 97–106, June 2002.

[34] J. Cong et al. Architecture-level synthesis for automatic interconnect pipelining. In *Proc. of the 41st Design Automation Conference*, pages 602–607, June 2004.

[35] L. Cordella et al. Performance Evaluation of the VF Graph Matching Algorithm. In *Proc. of the 1999 International Conference on Image Analysis and Processing*, volume 2, pages 1038–1041, 1999.

[36] Roberto Cordone, Fabrizio Ferrandi, Donatella Sciuto, and Roberto Wolfler Calvo. An efficient heuristic approach to solve the unate covering problem. In *Proc. of the 2000 Design, Automation and Test in Europe*, pages 364–371, 2000.

[37] Marc L. Corliss, E. Christopher Lewis, and Amir Roth. DISE: A programmable macro engine for customizing applications. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 362–373, 2003.

[38] J. Dehnert et al. The Transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proc. of the 2003 International Symposium on Code Generation and Optimization*, pages 15–24, March 2003.

[39] K. Ebcioglu and E. Altman. Daisy: Dynamic compilation for 100% architectural compatibility. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 26–38, June 1997.

[40] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for simd architectures with alignment constraints. In *Proc. of the SIGPLAN '04 Conference on Programming Language Design and Implementation*, pages 82–93, 2004.

[41] B. Fahs, T. Rafacz, S. Patel, and S. Lumetta. Continuous optimization. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 86–97. IEEE Computer Society, 2005.

[42] Brian Fahs et al. Performance characterization of a hardware mechanism for dynamic optimization. In *Proc. of the 34th Annual International Symposium on Microarchitecture*, pages 16–27, 2001.

[43] K. Fan, M. Kudlur, H. Park, and S. Mahlke. Cost sensitive modulo scheduling in a loop accelerator synthesis system. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, pages 219–230, November 2005.

[44] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981.

[45] D. Friendly, S. Patel, and Y. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *Proc. of the 25th Annual International Symposium on Computer Architecture*, pages 173–181, June 1998.

[46] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[47] S. Gochman et al. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 7(2):http://developer.intel.com/technology/itj/2003/volume07 issue02/, 2003.

[48] E.I. Goldberg, L.P. Carloni, T. Villa, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Negative thinking in branch-and-bound: the case of unate covering. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(3):281–294, March 2000.

[49] R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, March 2000.

[50] R. Govindarajan, E. R. Altman, and G. R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 85–94, November 1994.

[51] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, 2000.

[52] M. Gschwind. Instruction set selection for ASIP design. In *Proc. of the Seventh International Conference on Hardware/Software Codesign*, pages 7–11, 1999.

[53] Matthew Guthaus, Jeffrey Ringenberg, Dan Ernst, Todd Austin, Trevor Mudge, and Richard Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of the 4th IEEE Workshop on Workload Characterization*, pages 10–22, December 2001.

[54] J. R. Hauser and J. Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. In *Proc. of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 12–21, April 1997.

[55] K. Hazlewood and M. Smith. Generational cache management of code traces in dynamic optimization systems. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, pages 169–179, December 2003.

[56] B. Holmer. *Automatic Design of Computer Instruction Sets*. PhD thesis, University of California, Berkeley, 1993.

[57] Amir Hormati et al. Exploiting narrow accelerators with data-centric subgraph mapping. In *Proc. of the 2007 International Symposium on Code Generation and Optimization*, pages 341–353, March 2007.

[58] Ellis Horowitz and Sartaj Sahni. Exact and Approximate Algorithms for Scheduling Nonidentical Processors. *Journal of the ACM*, 23(2):317–327, 1976.

[59] Shiliang Hu, Ilhyun Kim, Mikko H. Lipasti, and James E. Smith. An approach for implementing efficient superscalar CISC processors. In *Proc. of the 12th International Symposium on High-Performance Computer Architecture*, pages 213–226, 2006.

[60] Shiliang Hu and James E. Smith. Using dynamic binary translation to fuse dependent instructions. In *Proc. of the 2004 International Symposium on Code Generation and Optimization*, pages 213–226, 2004.

[61] I. Huang and A. M. Despain. Synthesis of application specific instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(6):663–675, June 1995.

[62] Q. Jacobson and J. E. Smith. Instruction pre-processing in trace processors. In *Proc. of the 5th International Symposium on High-Performance Computer Architecture*, pages 125–133, 1999.

[63] G. Karypis and V. Kumar. *Metis: A Software Package for Paritioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Orderings of Sparce Matrices*. University of Minnesota, September 1998.

[64] R. Kastner et al. Instruction generation for hybrid reconfigurable systems. *ACM Transactions on Design Automation of Electronic Systems*, 7(4):605–627, April 2002.

[65] Vinod Kathail, Mike Schlansker, and Bob Rau. HPL-PD architecture specification: Version 1.1. Technical Report HPL-93-80(R.1), Hewlett-Packard Laboratories, February 2000.

[66] M. Mercaldi Kim, M. Mehrara, M. Oskin, and T. Austin. Architectural implications of brick and mortar silicon manufactoring. In *Proc. of the 34th Annual International Symposium on Computer Architecture*, page To Appear, 2007.

[67] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22(8):786–793, 1973.

[68] Israel Koren. *Computer Arithmetic Algorithms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[69] Andreas Krall and Sylvain Lelait. Compilation techniques for multimedia processors. *International Journal of Parallel Programming*, 28(4):347–361, 2000.

[70] Evgeny Krissinel and Kim Henrick. Common subgraph isomorphism detection by backtracking search. *Software: Practice and Experience*, 34(6):591–607, 2004.

[71] Krishna Kunchithapadam and James R. Larus. Using lightweight procedures to improve instruction cache performance. Technical Report CS-TR-1999-1390, January 1999.

[72] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *Proc. of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–327, 1988.

[73] Samuel Larsen, Rodric Rabbah, and Saman Amarasinghe. Exploiting vector parallelism in software pipelined loops. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, pages 119–129, 2005.

[74] Tanya Lattner. An Implementation of Swing Modulo Scheduling with Extensions for Superblocks. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, June 2005.

[75] C. Lee, M. Potkonjak, and W.H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, 1997.

[76] R. Leupers and P. Marwedel. Instruction selection for embedded DSPs with complex instructions. In *Proc. of the 1996 European Design Automation Conference*, pages 200–205, September 1996.

[77] Rainer Leupers and Peter Marwedel. Instruction-set modelling for asip code generation, 1996.

[78] John Levon. *OProfile - a System Profiler for Linux*, 2004. http://oprofile.sourceforge.net/doc/index.html, Retrieved June 6, 2007.

[79] S. Liao et al. Code optimization techniques for embedded DSP microprocessors. In *Proc. of the 32nd Design Automation Conference*, pages 599–604, 1995.

[80] S. Liao et al. Instruction selection using binate covering for code size optimization. In *Proc. of the 1995 International Conference on Computer Aided Design*, pages 393–399, 1995.

[81] Yuan Lin et al. Soda: A low-power architecture for software radio. In *Proc. of the 33rd Annual International Symposium on Computer Architecture*, pages 89–101, June 2006.

[82] J. Llosa et al. Swing modulo scheduling: A lifetime-sensitive approach. In *Proc. of the 5th International Conference on Parallel Architectures and Compilation Techniques*, pages 80–86, 1996.

[83] Josep Llosa, Mateo Valero, Eduard Ayguadé, and Antonio González. Hypernode reduction modulo scheduling. In *Proc. of the 28th Annual International Symposium on Microarchitecture*, pages 350–360, 1995.

[84] Gabriel H. Loh. Exploiting data-width locality to increase superscalar execution bandwidth. In *Proc. of the 35th Annual International Symposium on Microarchitecture*, pages 395–405, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[85] P. Lowney et al. The Multiflow Trace scheduling compiler. *Journal of Supercomputing*, 7(1):51–142, January 1993.

[86] Guangming Lu, Hartej Singh, Ming-Hau Lee, Nader Bagherzadeh, Fadi J. Kurdahi, and Eliseu M. Chaves Filho. The MorphoSys parallel reconfigurable system. In *Proc. of the 5th International Euro-Par Conference*, pages 727–734, 1999.

[87] Peter Marwedel and Gert Goossens. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, Boston, 1995.

[88] Binu Mathew and Al Davis. A loop accelerator for low power embedded VLIW processors. In *Proc. of the 2004 International Conference on on Hardware/Software Co-design and System Synthesis*, pages 6–11, 2004.

[89] Wen mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *Journal of Supercomputing*, 7(1):229–248, May 1993.

[90] Gokhan Memik, William H. Mangione-Smith, and Wendong Hu. NetBench: A benchmarking suite for network processors. In *Proc. of the 2001 International Conference on Computer Aided Design*, pages 39–42, 2001.

[91] Matthew Merten and Wen-Mei Hwu. Modulo schedule buffers. In *Proc. of the 34th Annual International Symposium on Microarchitecture*, pages 138 – 149, 2001.

[92] Paul Metzgen. A high performance 32-bit alu for programmable logic. In *Proc. of the 12th ACM Symposium on Field Programmable Gate Arrays*, pages 61–70, 2004.

[93] Takashi Miyamori and Kunle Olukotun. A quantitative analysis of reconfigurable coprocessors for multimedia applications. In *Proc. of the 6th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 2–11, 1998.

[94] R. Nair and M. Hopkins. Exploiting instruction level parallelism in processors by caching scheduled groups. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 13–25, June 1997.

[95] U. Nawathe et al. An 8-core, 64-thread, 64-bit, power efficient SPARC SoC (Niagara2), February 2007. In *Proc. of ISSCC*.

[96] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, pages 105–118, November 2005.

[97] Alex Pajuelo, Antonio Gonzlez, and Mateo Valero. Speculative dynamic vectorization. In *Proc. of the 29th Annual International Symposium on Computer Architecture*, pages 271–280, 2002.

[98] Krishna V. Palem, Surendranath Talla, and Weng-Fai Wong. Compiler Optimizations for Adaptive EPIC Processors. In *Proc. of the 2001 ACM Conference on Embedded Software*, pages 257–273, 2001.

[99] Sanjay J. Patel and Steven S. Lumetta. rePLay: A hardware framework for dynamic optimization. *IEEE Transactions on Computers*, 50(6):590–608, June 2001.

[100] Vlad Petric, Tingting Sha, and Amir Roth. Reno: A rename-based instruction optimizer. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 98–109, 2005.

[101] A. Peymandoust et al. Automatic instruction set extension and utilization for embedded processors. In *IEEE 14th International Conference on Application-specific Systems, Architectures and Processors*, pages 108–120, June 2003.

[102] J. Phillips and S. Vassiliadis. High-performance 3-1 interlock collapsing ALU's. *IEEE Transactions on Computers*, 43(3):257–268, 1994.

[103] J. Van Praet, G. Goosens, D. Lanner, H. De Man, and H. Synthesis. Instruction set definition and instruction selection for asip, 1994.

[104] D. Sreenivasa Rao and Fadi J. Kurdahi. On clustering for maximal regularity extraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(8):1198–1208, August 1993.

[105] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, November 1994.

[106] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. In *Proc. of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 283–299, June 1992.

[107] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai. Code generation for modulo scheduled loops. In *Proc. of the 25th Annual International Symposium on Microarchitecture*, pages 158–169, November 1992.

[108] B. R. Rau, D. W. L. Yen, and R. A. Towle. The cydra 5 departmental supercomputer. *IEEE Computer*, 1(22):12–34, January 1989.

[109] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable function units. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 172–180, December 1994.

[110] Hongbo Rong, Zhizhong Tang, R. Govindarajan, Alban Douillet, and Guang R. Gao. Single-dimension software pipelining for multidimensional loops. *ACM Transactions on Architecture and Code Optimization*, 4(1):7, 2007.

[111] P. Sassone and D. S. Wills. Dynamic strands: Collapsing speculative dependence chains for reducing pipeline communication. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 7–17, December 2004.

[112] Peter Sassone, D. Scott Wills, and Gabriel Loh. Static strands: safely collapsing dependence chains for increasing embedded power efficiency. In *Proc. of the 2005 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 127–136, June 2005.

[113] Yiannakis Sazeides, Stamatis Vassiliadis, and James E. Smith. The performance potential of data dependence speculation & collapsing. In *Proc. of the 29th Annual International Symposium on Microarchitecture*, pages 238–247. IEEE Computer Society, 1996.

[114] R. Schreiber et al. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing*, 31(2):127–142, 2002.

[115] David Seal. *ARM Architecture Reference Manual*. Addison-Wesley, London, UK, 2000.

[116] Mukund Sivaraman and Shail Aditya. Cycle-time aware architecture synthesis of custom hardware accelerators. In *Proc. of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 35–42, 2002.

[117] Francesco Spadini, Michael Fertig, and Sanjay J. Patel. Characterization of repeating dynamic code fragments. Technical Report CHRC-02-09, CHRC University of Illinois at Urbana-Champaign, September 2002.

[118] M.G. Stoodley and C.G.Lee. Software pipelining loops with conditional branches. In *Proc. of the 29th Annual International Symposium on Microarchitecture*, pages 262–273, December 1996.

[119] F. Sun et al. Synthesis of custom processors based on extensible platforms. In *Proc. of the 2002 International Conference on Computer Aided Design*, pages 641–648, November 2002.

[120] Tensilica Inc. *Xtensa Architecture and Performance*, September 2002. http://www.tensilica.com/xtensa_arch_white_paper.pdf.

[121] Trimaran. An infrastructure for research in ILP, 2000. http://www.trimaran.org/.

[122] J. R. Ullman. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

[123] Sriram Vajapeyam, P. J. Joseph, and Tulika Mitra. Dynamic vectorization: A mechanism for exploiting far-flung ILP in ordinary programs. In *Proc. of the 26th Annual International Symposium on Computer Architecture*, pages 16–27, 1999.

[124] Scott J. Weber and Kurt Keutzer. Using minimal minterms to represent programmability. In *Proc. of the 2005 International Conference on on Hardware/Software Codesign and System Synthesis*, pages 63–68, 2005.

[125] M. J. Wirthlin and B. L. Hutchings. DISC: The dynamic instruction set computer. In *Proc. of the 1995 Field Programmable Gate Arrays for Fast Board Development and Reconfigurable Computing*, pages 92–103, 1995.

[126] L. Wu, C. Weaver, and T. Austin. Cryptomaniac: A fast flexible architecture for secure communication. In *Proc. of the 28th Annual International Symposium on Computer Architecture*, pages 110–119, June 2001.

[127] Peng Wu, Alexandre E. Eichenberger, and Amy Wang. Efficient simd code generation for runtime alignment and length conversion. In *Proc. of the 2005 International Symposium on Code Generation and Optimization*, pages 153–164, 2005.

[128] Zhi Alex Ye et al. CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proc. of the 27th Annual International Symposium on Computer Architecture*, pages 225–235, 2000.

[129] Sami Yehia et al. Exploring the design space of LUT-based transparent accelerators. In *Proc. of the 2005 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 11–21, September 2005.

[130] Sami Yehia and Olivier Temam. From sequences of dependent instructions to functions: An approach for improving performance without ILP or speculation. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, pages 238–249, June 2004.

[131] P. Yu and T. Mitra. Characterizing embedded applications for instruction-set extensible processors. In *Proc. of the 41st Design Automation Conference*, pages 723–728, June 2004.