UNIVERSITY OF CALIFORNIA,
IRVINE

Compiler-in-the-Loop Exploration of Programmable Embedded Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Aviral Shrivastava

2006

The dissertation of Aviral Shrivastava
is approved and is acceptable in quality
and form for publication on microfilm:

Committee Chair

University of California, Irvine
2006

To my parents.

Contents

List of Figures

List of Tables

Acknowledgments

Foremost, I would like to thank my advisor, Prof. Nikil Dutt, for his guidance and support during my docotrate. Prof. Dutt gave me ample freedom and space to explore and test my ideas. His infinite patience with me, even when I was disturbed due to lack of progress gave me a lot of confidence in myself and my work. Over the years, I have grown to appreciate his approach and vision of a Ph.D.

I would also like to thank Prof. Alex Nicolau and Prof. Alex Veidenbaum for their guidance and for serving on my thesis committee. I would like to thank Prof. Tony Givargis for lending me an ear and a lot of encouragment, especially towards the end of my Ph.D.

I would like to thank Eugene Earlie for being my mentor. Thank you Eugene for taking the time to discuss my work at length. Your confidence in me and my work was a constant source of encouragement throughout my Ph.D. process.

I would like to thank Melanie Sanders for being so forthcoming and going out of her way and helping us throughout the years. Without you, life would at the least be very tough.

I would also like to thank my lab-mates in the ACES lab at the Center for Embedded Computer Systems (CECS); without them this process would have been much difficult. In particular, I would like to thank Partha Biswas, Ashok Halambi, Ilya Issenin, Mahesh Mamidipaka and Vivek Sinha. My heartfelt thanks to the other members at CECS who have provided friendship, support and stimulating work environment.

I would like to my wife, Indu for her unflinching support and patience. She is becoming ever more increasingly important part of my life. She has been with me in all the crests and troughs of this Ph.D. process. She is an important source of courage in tough times.

I would like to thank my parents to whom this thesis is dedicated. It is due to the vision of my father Shri Shiv Swaroop Shrivastava and mother Smt. Munni Devi Shrivastava, their personal sacrifices in providing me the best possible education and healthy environment at home that I have reached here. I want to thank them for dreaming with me and being a constant source of inspiration. My accomplishment is in a way their accomplishment as well.

I would also like to thank my sister Ms. Vandita Shrivastava for her amazing amount of affection for me. She always cheers me up, rejuvinates me and puts me back on track with full steam.

Curriculum Vitae

Aviral Shrivastava

Education

2006	**Ph.D.** in Computer Science, University of California, Irvine
2002	**M.S.** in Computer Science, University of California, Irvine
1999	**B.Tech.** in Computer Science and Engineering, IIT Delhi, India

Summary of Work and Research experience

June 2000 – Present	Graduate RA	University of California, Irvine
June 2003-December 2003	Research Intern	Strategic CAD Labs
June 2002-August 2002	Research Intern	PICO Group, HP Labs
September 1999-June 2000	CAD Engineer	Philips Semiconductors, Netherlands
May 1999-August 1999	Research Associate	IIT Delhi, India

Publications

Software

[S1] "PBExplore: A Compiler-in-the-Loop Design Space Exploration Framework for Partially Bypassed Processors",
http://www.ics.uci.edu/~aviral/pbexplore

[S2] "EXPRESSION: An Architecture Description Language based Retargetable Compiler-Simulator toolkit",
http://www.ics.uci.edu/~express

Journals

[J1] "Compilation Framework for Code Size Reduction using Reduced Bit-width ISAs", Aviral Shrivastava, Partha Biswas, Ashok Halambi, Nikil Dutt, and Alex Nicolau, *IEEE Transactions on Design Automation of Electronic Systems*, January 2006.

[J2] "Operation Tables for Processors with Partial Bypasses", Aviral Shrivastava, Eugene Earlie, Nikil Dutt, and Alex Nicolau, *IEEE Transactions on Very Large Scale Integrated Circuits*, Accepted for publication.

[J3] "ADL-driven Software Toolkit Generation for Architectural Exploration of Programmable SOCs", Prabhat Mishra, Aviral Shrivastava and Nikil Dutt, *ACM Transactions on Design Automation of Electronic Systems*, Accepted for publication.

Conferences:

[C1] "Automatic Generation of Operation Tables for Fast Exploration of Bypasses in Embedded Processors", Sanghyun Park, Aviral Shrivastava, Nikil Dutt, Alex Nicolau, Eugene Earlie, and Yunheung Paek, *Proceedings of the International Conference on Design Automation and Test in Europe*, 2006

[C2] "Compilation Techniques for Energy Reduction in Horizontally Partitioned Cache Architectures", Aviral Shrivastava, Ilya Issenin and Nikil Dutt, *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems on Hardware/Software Codesign and System Synthesis*, 2005

[C3] "Aggregating Processor Free Time for Energy Reduction", Aviral Shrivastava, Eugene Earlie, Nikil Dutt, and Alex Nicolau, *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2005

[C4] "PBExplore: A Framework for Compiler-in-the-Loop Exploration of Partial Bypassing in Embedded Processors", Aviral Shrivastava, Nikil Dutt, Alex Nicolau and Eugene Earlie, *Design, Automation and Test in Europe*, 2005

[C5] "Operation Tables for Scheduling in the Presence of Incomplete Bypassing", Aviral Shrivastava, Eugene Earlie, Nikil Dutt and Alex Nicolau, *International Conference on Hardware/Software Codesign and Software Synthesis*, 2004

[C6] "Energy Efficient Code Generation using rISA", Aviral Shrivastava and Nikil Dutt, *Asia South Pacific Design Automation Conference*, 2004

[C7] "A Design Space Exploration Framework for Reduced Bit-width Instruction Set Architecture (rISA) Design", Ashok Halambi, Aviral Shrivastava, Partha Biswas, Nikil Dutt and Alex Nicolau, *International Symposium on System Synthesis*, 2002

[C8] "A Customizable Compiler Framework for Embedded Systems", Ashok Halambi, Aviral Shrivastava, Nikil Dutt and Alex Nicolau, *International Workshop on Software and Compilers for Embedded Systems*, 2001

[C9] "Hardware-Software Partitioning of Concurrent Sequence Flow Graphs", Aviral Shrivastava, Mohit Kumar, Sanjiv Kapoor, Shashi Kumar and M. Balakrishnan, *International Conference on VLSI Design*, 2000

Design

[D1] "An optimized design of a 32-bit Floating Point Multiplier", Aviral Shrivastava, Pankaj Bharti, *International Conference on VLSI Design*, 1999

Abstract of the Dissertation

Compiler-in-the-Loop Exploration of
Programmable Embedded Systems

by
Aviral Shrivastava
Doctor of Philosophy in Information and Computer Science
University of California, Irvine, 2006
Professor Nikil D. Dutt, Chair

Increasing complexity of embedded systems, and shortening time-to-market makes designer productivity the key concern in embedded system development. As a result Programmable Embedded Systems — that have a programmable processor and memory subsystem to support the software part of the application — are becoming an attractive platform for embedded system design. Programmable embedded systems greatly enhance design reuse, reduce the complexity, and time-to-market via software. The application-specific, strict, multi-dimensional design constraints, result in embedded processor designs being *highly customized*. Embedded processors often feature design idiosyncracies, custom-algorithms, and sometimes even miss some architectural features. Consequently, code generation for the embedded processors is a challenging task. However, if the compiler is able to exploit the architectural features of the embedded processors, it can make a tremendous difference in the power, performance, etc. of the eventual system. Existing embedded system design/exploration techniques either do not consider compiler effects on the design, or include the compiler effects in an ad-hoc manner, which may lead to inaccurate evaluation of design choices and therefore result in suboptimal design decisions. This thesis proposes a *Compiler-in-the-Loop Exploration* approach, – a systematic method to include compiler effects during architectural evaluation of embedded systems. This dissertation demonstrates the need and usefulness of the proposed methodology at several levels of embedded system design abstraction: at the the instruction set architecture level, at the processor pipeline design level, at the memory design level, and at the processor-memory interface level. At each level of design abstraction, this dissertation demonstrates that the proposed methodology results in a more meaningful exploration of design space leading to better design decisions with respect to the design goals of performance, code size, energy and power consumption.

Chapter 1

Introduction

1.1 Embedded Systems

An embedded system is a special purpose computer system, which is encapsulated by the device it controls. The embedded system takes inputs from a pre-defined, a rather restrictive interface, and has a very specific predefined functionality, unlike a general-purpose computer system. For example, an embedded system is used in most modern cars to monitor the engine emissions and adjust the engine operations to keep emissions as low as possible. This system receives inputs from a variety of sensors, e.g., oxygen sensor, air pressure sensor, air temperature sensor, engine temperature sensor, throttle position sensor, knock sensor, etc. The embedded system uses this information to control fuel injectors, spark plugs, idle speed etc. to achieve the best engine performance while minimizing the engine emissions.

1.1.1 Ubiquitous and Pervasive

Our everyday lives are becoming increasingly dependent on Embedded Systems. Figure 1.1 depicts that embedded systems play an important role in most aspects of our life. To start with we may have some embedded systems inside our body, e.g., a rhythm generator. Then, we might be wearing a few embedded systems on our body, e.g., watch and music players. We interact with embedded systems all the time; at home, microwave, refrigerator, home security system, are embedded systems. At office, the printer and fax machines are examples of embedded systems. Even while going to office, we are dependent on over 50 embedded systems present in modern-day cars, and even more at each traffic control light. Most entertainment systems are examples of high-end embedded systems. In fact even while

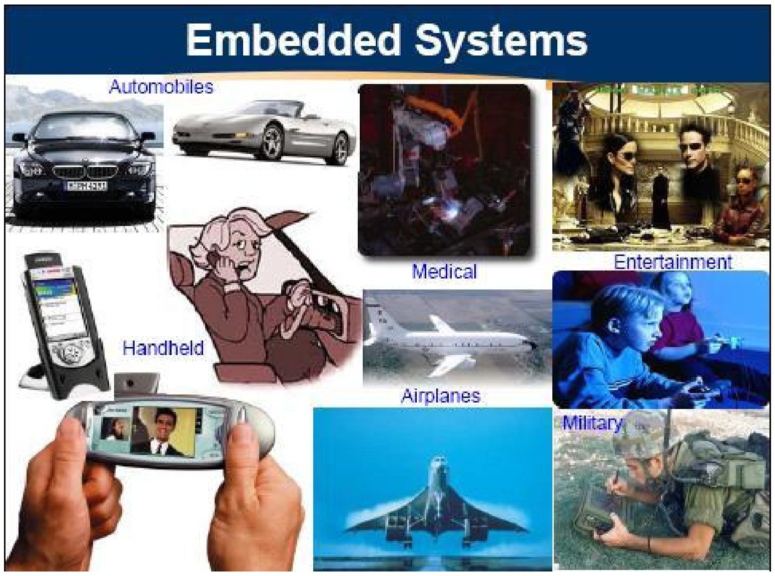


Figure 1.1: Embedded System Collage, courtesy Prof. Prabhat Mishra

we are sleeping, the alarm radio clock is an embedded system we are dependent upon to wake up on time.

While some embedded systems merely assist us to communicate with other people, e.g., mobile phones, others help to keep our lives organized, e.g., digital diary, PDA, there are yet other classes of embedded systems on which we even depend on for our lives. For example, several embedded systems, like the ones in automotive devices and controls, railways, aircraft, aerospace and medical devices are safety-critical applications.

Embedded systems have wide-ranging impact on society, including security, privacy and modes of working and living. Embedded Systems may not be visible to the customer as "computers" in the ordinary sense. New processors and methods of processing, sensors, actuators, communications and infrastructures are "enablers" for this very pervasive computing. They are in a sense ubiquitous, that is, almost invisible to the user and almost omnipresent. As such, they form the basis for a significant economic growth.

1.1.2 Impact of Embedded Systems

More than 98% of the processors designed today end up in embedded systems. According to a report, "Future of Embedded Systems Technology" from the BCC Research Group, the worldwide embedded systems market was estimated at $45.9 billion in 2004. Expected to grow at an average annual growth rate (AAGR) of 14% over the next six years, this market will cross $90 billion before 2010. To accommodate this exponential growth chip designers and manufacturers have constantly pushed the envelope of technological, physical, and design constraints. Various innovations and paradigm-defining ideas have taken shape as a result.

1.2 Constraints on Embedded Systems

Embedded computer systems constitute the widest possible use of computer systems; it includes all computers other than those specifically intended as general-purpose computers. The diversity in the application of embedded systems translate into a very diverse set of stringent application-specific, multi-dimensional constraints.

1.2.1 Application-Specific Constraints

Embedded systems are characterized by providing a set of function, or functions, that is not itself a computer in the ordinary sense. Ranging from a portable music player, to real-time control of systems, like of the space shuttle, the functionality and environment of embedded systems varies a lot. For example, the embedded system used in space shuttles, or oil well borehole must be able to operate at an unusual temperature and pressure, and UV radiation ranges. The embedded system that controls the braking mechanism in cars should respond quickly every time. Such embedded systems have real-time constraints on their functionality.

1.2.2 Multi-dimensional Constraints

One distinguishing characteristic of embedded systems is the multi-dimensional constraints under which these systems must operate. A few of these constraints are:

Performance Embedded systems, like all other computer systems have performance constraints, and it is not surprising that more and more performance in being desired by these embedded systems.

Cost A very common and important constraint for embedded systems is the cost. In the domain of high volume embedded systems, e.g., a portable music player, reducing cost becomes a major concern. These systems will often have just a few chips, a highly integrated CPU, a custom chip that controls all other functions and a single memory chip. In these designs each component is selected and designed to minimize system cost.

Weight Increased weight of hand-held embedded systems can render then unusable, or atleast unsellable. For example the lightest mobile phones can be sold at premium pricing. Automobile industry prefers light components, because weight directly impacts the fuel consumption of the cars. Same is the case with embedded systems deployed in space or air crafts.

Power/Energy Of all the constraints applicable on embedded systems, energy, or power constraints may well be the most important ones, especially for battery operated embedded systems. A strict limitation on the power consumption of the embedded system comes from the power capacity of the battery. Furthermore, increase in energy consumption results in either larger (and heavier) battery, or frequent charging, both of which are undesirable.

Real-time Many embedded systems have real-time system constraints that must be met. The braking mechanism in cars should respond quickly and reliably every time. Such embedded systems have real-time constraints on their functionality. There needs to be an acceptable limit on the theoretical limit on the time when the brake pedal is pressed to the time when the brakes are applied. These hard timing constraints are modeled as real-time constraints that must be met in the system.

Time-to-Market The life-time of most consumer electronic embedded systems is decreasing drastically. Even time the time of inception of an embedded system device, it is well known that the window in which the embedded system should be brought into the market is very short. The deadlines to market the embedded systems are very short, and the profit therein is very sensitive to it.

Size Embedded systems, especially the ones that go inside the body have very strict size constraints. More broadly size constraint, or form factor is an important constraint in cell phones, PDAs too.

Thus, unlike general purpose computers, embedded systems have constraints in many dimensions, e.g., cost, area, power consumption, time-to-market, reusability, and even weight of the embedded system.

1.2.3 Stringent Constraints

Embedded systems are much more sensitive to multi-dimensional constraints than general purpose systems. For example even if the weight, or area, or power consumption of a general purpose computer is a little more than expected, it is okay, because the general purpose computer is not supposed to be frequently moved, and is attached to the power plug in the wall. However increase in the power consumption, or weight of a PDA may render it unsellable, and unuseable. If the remote sensing applications consume more power than it can be generated by the solar panels, then they cannot even function.

To summarize, embedded systems are characterized by *strict multi-dimensional design constraints.*

1.3 Designer Productivity

1.3.1 Increase in Complexity

A very clear and important trend in embedded systems is the continuous increase in complexity. More and more functionality is desired and delivered in the next generation of electronic consumer systems. Together with the convergence of devices, this has led to a steep increase in the complexity of embedded systems. For instance Handspring technologies makes a popular cell phone series Treo. In 2002, Treo 300 was launched, equipped with a 33 MHz processor. The next version of Treo was launched in 2004, Treo 600, and it has a 133 MHz processor. Cell phone processors are fast increasing in the capabilities, and are fast marching towards the 600-megahertz range. Although evolving communications standards are a reason for increasing performance requirements of cell phones, another important reason is the convergence of functionality in cell phone. Call processing is now just one small aspect of modern cell phones. They already come loaded with Palm organizer software and a Blazer Web browser, and can run Microsoft Outlook, Word, Excel, and other core business applications. Users can read and send e-mail, view PDFs, inspect and make changes to documents, review change orders, and even pull up drawings to inspect with co-workers. They can also call the office to check voice mail. The cell phones and PDAs are already integrated, the vision is now to replace the laptop by these hybrid devices. For example, Siemens is developing the SX-1, a phone that uses a laser to project a virtual full-size keyboard onto a flat surface, to make the laptops obsolete. Similarly there are increasing performance demands for network routers, video games, and avionic software.

1.3.2 Shrinking Time-to-Market

The time elapsed from the conception of a product to when it is launched is called Time To Market (TTM), and is one of the important factors which determines the profits for the product. The embedded system market is characterized by amazingly small product time to market window. Computers seem obsolete a few months after rollout and new wireless phones come out two weeks after you bought one. In such a world, the TTM can make or break an equipment manufacturer.

1.4 Programmable Embedded Systems

Increasing complex system functionalities and time-to-market pressures are changing how electronic systems are designed and implemented today. The escalating nonrecurring engineering (NRE) costs to design and manufacture the chips have tiled the balance towards achieving greater design reuse. As a result hardwired application specific integrated circuit (ASIC) solutions are no longer attractive. Increasingly we are seeing a shift toward systems implemented on programmable platforms. Embedded systems implemented on such programmable platforms are called "Programmable Embedded Systems". Programmable embedded systems are an attractive option for modern embedded systems as they provide easier and faster implementation, easier "reusability" and "upgrade-ability" via software.

When time-to-market is the major concern, designers definitely prefer to use "programmable" components instead of custom hardware components in the embedded systems. In the programmable embedded system as shown in Figure 1.2, the processor and the memory are referred to as the *programmable components*. For example it is preferable to implement the user interface in the phone in software. This allows cheap and fast upgradeability of the user-interface of the phone. In fact the main reason for implementing some functionality in custom-made hardware is to meet the speed and power constraints. With increasing speeds and low-power processor technology available, more and more functionality is being shifted to the software. However by doing this, the onus of developing the

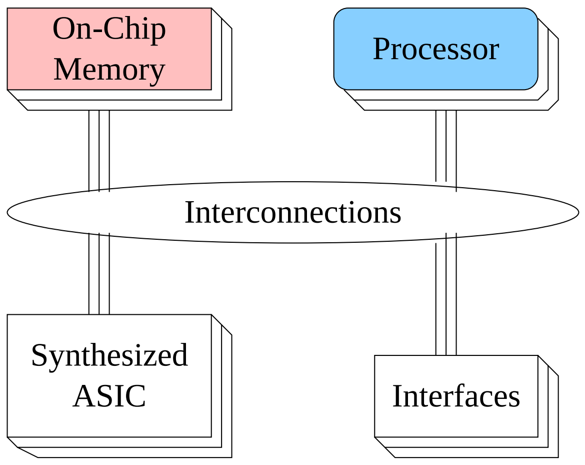


Figure 1.2: Programmable Embedded System

system is shifted on to the software engineer.

1.4.1 Highly Customized Architectures

The programmable component in the embedded system (or the embedded processor) is designed very much like the general purpose processors, but is more specialized and customized to the application domain. To meet the strict multi-dimensional constraints applicable on the embedded system, *customization* is very important. For example, even though register renaming increases performance in processors by avoiding false data dependencies, embedded processors may not be able to employ it because of the high power consumption and the complexity of the logic. Therefore embedded processors might deploy a “trimmed-down” or “light-weight” version of register renaming, which provides them the best compromise on the important design parameters.

In addition designers often implement some *irregular design features*, which are not common in general purpose processors, but will lead to significant improvements in some design parameters for the relevant set of applications. For example, several cryptography application processors come with hardware accelerators that implement the complex cryptography algorithm in the hardware. By doing so, the cryptography applications can be made faster, and consume less power, but may not have any noticeable impact on normal applications. Embedded processor architectures often have such application specific “idiosyncratic” architectural features.

And last but not the least, some design features that are present in the general purpose processors may be entirely missing in embedded processors. For example, support for prefetching is now a standard feature in general purpose processors, but it may consume too much energy and require too much extra hardware to be appropriate in an embedded processor.

1.4.2 Compilation Challenges

High levels of customization and the presence of idiosyncratic design features in embedded processors, leaves the compiler for the embedded processors in a very tough spot. Compilers for general purpose processors are not suitable for embedded processors for several reasons as detailed in this section.

1.4.2.1 Different ISA

Typically embedded processors have different instruction set architectures (ISA) than general purpose processors. While IA32, and PowerPC are the most popular ISAs in the general purpose processors, ARM and MIPS are the most popular instruction sets in embedded processors. The primary reason for the difference in ISAs is because embedded processors are often built ground up to optimize for their design constraints. For instance the ARM instruction set results in very small code size. Compiling an application in ARM code may often lead to the minimum code size.

1.4.2.2 Different Optimization Goals

Even if the compilers can be modified to compile for a different instruction set, the optimization goals of the compilers for general purpose processors and embedded processors differ a lot. Most general purpose compiler technology aims towards high performance and less compile-time. However for many embedded systems, code size may be a more important design goal, because the binary image needs to fit in the limited memory present in the embedded system. Furthermore power consumption is a very important goal for the compilers of embedded processors, which may not be a concern for the compilers for general purpose processors.

1.4.2.3 Limited Compiler Technology

Even though techniques may be present to exploit the regular design features in the general purpose processors, compiler technology to exploit the "customized" version of the architectural technique may be absent. For example, predication is a standard architectural feature employed in most high-end processors. In predication, the execution of each instruction is conditional on the value of a bit in processor state register, called condition bit. The condition bit can be set by some instructions. Predication allows a dynamic decision about whether to execute an instruction or not. However, due to the architectural overhead of implementing predication, sometimes a very limited support for predication is deployed in embedded processors. For example, in the Starcore architecture [1], there is no condition bit, there is just a special conditional move instruction (e.g., *cond_move R1 R2, R3 R4*), whose semantics are: if (R1 > 0) *move R1 R3*, else *move R1 R4*. To achieve the same effect as predication, the computations should be performed locally, and then the conditional instruction can be used to dynamically decide to commit the result or not. Now in such cases, the existing techniques and heuristics developed for predication do not work. New techniques have to be developed to exploit this "flavor" of predication in the architecture. The first challenge in developing compilers for embedded processors is therefore to enhance the compiler technology to exploit novel and idiosyncratic architectural features present in embedded processors.

1.4.2.4 Limited Resources

Since the architectural feature deployed in embedded processor is irregular, idiosyncratic or customized, the analysis required to exploit the feature is typically more complex. However, since the analysis might have to run on the embedded processor itself, it may not be a feasible solution. Therefore, merely developing a compilation technique to exploit an architectural feature in the embedded processor is not enough, a light-weight analysis is required that can be employed in the resource constrained embedded processor. For example, register renaming is a popular architectural feature present in many high-end processors. It takes care of all the false dependencies in the code, and therefore does not need any fine grain compiler support. However due to the high complexity of register renaming, it is not typically present in embedded processors. Therefore there is scope of developing fine grain compiler scheduling mechanisms that can improve the performance by avoid processor pipeline stalls due to data hazards for embedded processors. Furthermore, some embedded processors may even deploy partial register renaming. In such customized cases, the analysis to detect the data hazards becomes very complicated, as it has to be done on a case-to-case basis; as a result, even if such a technique is developed, owing to it's complexity, it may be unusable in the embedded system. Therefore a major challenge is generating code for embedded processors is to develop "light-weight" techniques to exploit the irregular, idiosyncratic, design features in the embedded processor.

1.4.2.5 Avoid penalty due to missing design features

On the same lines, several times embedded systems simply omit some architectural features that are common in general purpose processors. For example the support for prefetching may be absent in embedded processor. In such cases, the challenge is to minimize the power and performance loss due to the missing architectural feature.

To summarize, code generation for embedded processor is very challenging due to their non-regular architectures, and stringent multidimensional constraints on them.

1.4.3 Compilation is Effective for Embedded Processors

However, the brighter side of this story is that it has been shown time and again that when a compilation technique is developed to exploit an "idiosyncratic" architectural features of embedded systems, the power, performance, etc. of the system can be significantly improved. For example, a compiler technique to support the partial predication can achieve almost the same performance as complete predication [2]. Therefore it is definitely very important to investigate compiler techniques to exploit the various architectural features present in embedded systems. This is one of the most important goals of this thesis. In this thesis, we propose several compilation techniques to exploit the architectural features present in embedded processors.

1.5 Compiler-Assisted Embedded Processor Design

Given the significance of the compiler on the processor power and performance, it is only logical that the compiler must play an important role in embedded processor design. This is especially important for embedded systems.

1.5.1 Traditional Design Space Exploration

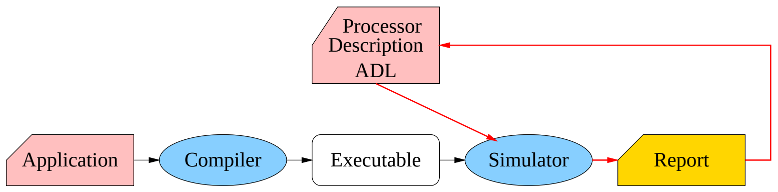


Figure 1.3: Traditional Simulation-Only Design Space Exploration

Figure 1.3 models the traditional design methodology for exploring prcoessor architectures. In the traditional approach, the application is compiled once to generate an executable. The executable is then simulated over various architectures to choose the best architecture. We call such traditional design methodology as *Simulation-Only* (SO) Design Space Exploration (DSE). The SO DSE of embedded systems does not incorporate compiler effects on the embedded processor design. However the compiler effects on the eventual power, performance characteristics can be incorporated in embedded processor design in an ad-hoc manner in the existing methodology. For example, the hand-generated code can be used to reflect the code that the actual compiler will eventually generate. This hand-generated code can be used to evaluate the architecture. However, such a scheme may be erroneous and result in sub-optimal design decisions. A systematic way to incorporate compiler hints while designing the embedded processor is needed.

1.5.2 Compiler-in-the-Loop Exploration

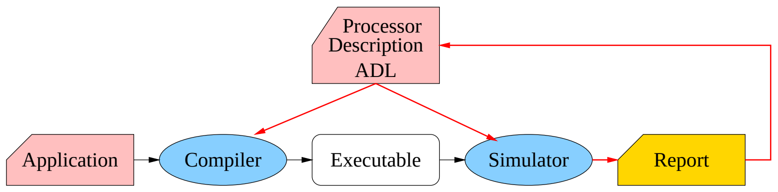


Figure 1.4: Compiler-in-the-Loop Design Space Exploration

In this thesis, we propose a systematic method to incorporate compiler effects during the embedded processor design. Figure 1.4 describes our proposed Compiler-Assisted, or Compiler-in-the-Loop (CIL) schema for DSE. In this scheme, for each architectural variation, the application is compiled (using an architecture-sensitive compiler), and the executable is simulated on a simulator of the architectural variation. Thus the evaluation of the architecture incorporates the compiler effects in a systematic manner.

The key enabler of the CIL DSE methodology is a architecture-sensitive compiler. While a conventional compiler takes only the application as input and generates the executable, an architecture-sensitive compiler also takes the processor architecture description as an input. The architecture-sensitive compiler exploits architectural features present in the described system, and generates code for the specific architecture configuration.

1.6 Thesis Contributions

This thesis advances both the compiler technology and processor design technology in synergistic ways. This thesis advances compiler technology by developing novel techniques to exploit architectural features present in embedded processors. Additionally, this thesis demonstrates the need and usefulness of CIL DSE to effectively and accurately explore the embedded processor design space.

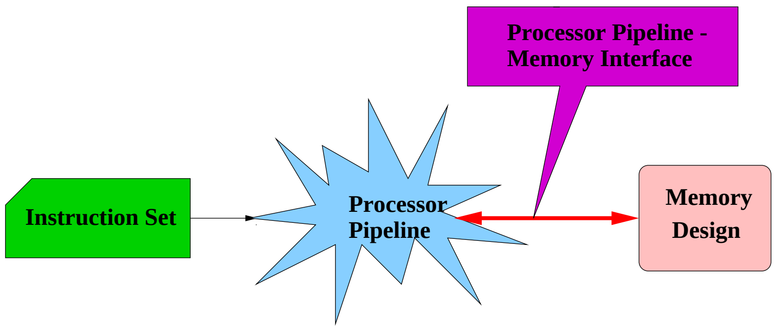


Figure 1.5: Processor Design Abstractions

This dissertation develops novel compilation techniques to exploit architectural features present in embedded processors and demonstrate the need and usefulness of CIL DSE at several abstractions of processor design, as shown in Figure 1.5: at the processor instruction set design abstraction, at the processor pipeline design abstraction, at the memory design abstraction, and at the processor memory interaction abstraction.

At the processor pipeline design abstraction, in Chapter 2, we first develop a novel compilation technique for generating code for processors with partial bypassing. Partial bypassing is a popular microarchitectural feature present in embedded systems because although full bypassing is the best for performance, it may have significant area, power and wiring complexity overheads. However, partial bypassing in processors poses a challenge for compilers, as there are no techniques to accurately detect pipeline hazards in a partially bypassed processors. Our Operation Table based modeling of the processor allows us to accurately detect all kinds of pipeline hazards, and generates up to 20% better performing code than a bypass insensitive compiler.

During processor design, the decision to add/remove a bypass is typically made by designer's intuition and/or SO DSE. However, since the compiler has significant impact on the code generated for a bypass configuration, the SO DSE may be significantly inaccurate. The comparison of our CIL with SO DSE demonstrates that not only do these two explorations result in significantly different evaluations of each bypass configurations, but they also exhibit different trends for the goodness of bypass configurations. Consequently, the traditional SO DSE can result in sub-optimal design decisions, justifying the need and usefulness of our CIL DSE of bypasses in embedded systems.

At the instruction set design abstraction, in Chapter 3, we first develop a novel compilation technique to generate code to exploit *reduced bit-width Instruction Set Architectures* (rISA). rISA is a popular architectural feature in which the processor supports two instruction sets. The first instruction set comprises instructions which are 32-bits wide, and the second is a narrow instruction set which comprises 16-bit wide instructions. rISA architectures were originally conceived to reduce the code size of the application. If the application can be expressed in the narrow instructions only, then upto 50% code compression can be achieved. However since the narrow instructions are only 16-bits wide, they implement limited functionality, and can access only a small subset of the architectural registers. Our register pressure heuristic consistently achieves 35% code compression as compared to 14% achieved by existing techniques.

In addition we also find out that the code compression achieved is very sensitive on the narrow instruction set chosen and the compiler. Therefore during processor design, the narrow instruction set should be designed very carefully. We employ our CIL DSE technique to design the narrow instruction set. We find that correctly designing the narrow instruction set can double the achievable code compression.

At the processor pipeline - memory interface design abstraction, in Chapter 4, we first develop a compilation technique to aggregate the processor activity and therefore reduce the power consumption when the processor is stalled. Fast and high bandwidth memory buses — although best for performance — can have very high cost, energy consumption, and design complexity. As a result embedded processors often employ slow buses. Reducing the speed of the memory bus increases the time a processor is stalled. Since the energy consumption of the processor is lower in the stalled state, the power consumption of the processor decreases. However there is further scope for power reduction of the processor by switching the processor to IDLE state while it is stalled. However, switching the state of the processor takes 180 processor cycles in the Intel XScale, while the largest stall duration observed in the *qsort* benchmark of the MiBench suite is less than 100 processor cycles. Therefore it is not possible to switch the processor to low power IDLE state during naturally occurring stalls during the application execution. Our technique aggregates the memory stalls of a processor into a large enough stall, so that the processor can be switched to the low power IDLE state. Our technique is able to aggregate up to 50,000 stall cycles, and by switching the processor to low power IDLE state, the power consumption of the processor can be reduced by up to 18%.

There is a significant difference in the estimation of the processor power consumption between the SO DSE and CIL DSE. SO DSE can significantly overestimate the processor power consumption for a given memory bus configuration. This bolsters the need and usefulness of including compiler effects during the exploration and therefore highlights the need for CIL DSE.

At the memory design abstraction in Chapter 5, we first develop a novel compilation technique to optimize for energy consumption in Horizontally Partitioned Cache (HPC) architectures. HPC is a popular memory architectural feature present in embedded systems in which the processors have multiple (typically two) caches at the same level of memory hierarchy. Wisely partitioning data between the caches can result in performance and energy improvements. However existing techniques target at performance improvements and achieve energy reduction only as a byproduct. Our energy oriented data partitioning technique is able to reduce the energy consumption by the memory subsystem by 50%, while losing 3% on performance.

We observe that the energy reduction obtained using HPCs is very sensitive on the caches sizes. Therefore it is important to include compiler effects while deciding the caches sizes. As compared to SO DSE of HPC configurations, CIL DSE results in discovering HPC configurations which result in 33% less energy consumption.

Finally we summarize and conclude our thesis in Chapter 6.

Chapter 2

Processor Pipeline Design

The processor pipeline design is arguably the most important aspect of processor design. Design of the processor pipeline includes deciding the length and the width of the processor, the register renaming and instruction scheduling policy, the number of integer, and floating point ALUs, multipliers, load store units, branch units and much more. As depicted in Figure 2.1, the processor pipeline design also heavily depends on and influences the instruction set architecture and the memory design of the processor.

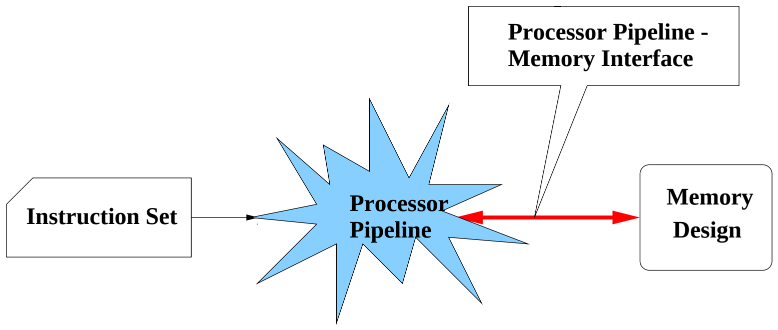


Figure 2.1: Processor Pipeline Design Abstraction

Typically the next generation of a processor has the same instruction set architecture, but has a different processor pipeline structure. For example, the pipelines implementing the ARM instruction have evolved from a simple 3-stage pipeline to a 5-stage pipeline in StrongARM 11000 and currently to the 7-stage pipeline in the Intel XScale. The processor pipeline is the chief means of achieving the power, performance objectives of the processor.

In the context of embedded systems, there is a tremendous opportunity to tune the application code to the processor pipeline features to optimize for power and performance etc. Consequently when the processor pipeline is modified (upgrade a processor), a new compiler is released for the new processor. Note that since the ARM instruction sets are backward compatible, the application code that was executing on the old ARM processor will execute correctly on the new processor, although it will not be executing efficiently. The compiler for the new processor tries to exploit the processor pipeline features and achieve efficient execution of the application code.

However, it takes a very long time to develop the compiler for the new generation of the processor. This is primarily due to very limited reuse of the compiler across the processor designs. The heuristics for speedup, or low-power employed in the previous version of the processor do not work for the new processor; as a result, significant parts of the compiler have to re-written and re-tuned. Retargetable compiler technology aims to solve this problem by developing a compiler, in which the heuristics and compiler passes are parameterized on the architectural features. In the ADL-based retargetable compilers, the processor pipeline is specified in an Architecture Description Language (ADL), and the compiler is parameterized on the architecture description.

Retargetable compilation techniques have been developed to exploit the length, and width of the processor. Retargetable compilation techniques have been developed in the EXPRESSION project [3], which allows the designers to add/remove a pipeline stage and generate efficient code for the processor pipeline design. The Trimaran [4] compiler uses the MDes ADL to produce code for parameterizable EPIC architectures. Thus designers can accurately explore the impacts of the modifying the processor pipeline length, while including the compiler effects. What is more important is that emission of such compiler effects not including the compiler impacts results in significant error in the judgement of the goodness of the pipeline designs.

In this chapter we focus on developing a retargetable compilation technique for partially bypassed processors and demonstrate the effectiveness of CIL DSE exploration methodology to design the bypasses of the processor.

2.1 Partial Bypassing

Bypasses, or forwarding paths are simple yet powerful and widely used feature in modern processors to eliminate some data hazards [5]. With Bypasses, additional datapaths and control logic are added to the processor so that the result of an operation is available for subsequent dependent operations even before it is written in the register file. However, this benefit of bypassing is accompanied by significant impact on the wiring area on the chip, possibly widening the pitch of the execution-unit datapaths. Datapaths including the bypasses often are timing critical and cause pressure on cycle time, especially the single cycle paths. The delay of bypass logic can be significant for wide issue machines. Due to extensive bypassing very wide multiplexors or buses with several drivers may be needed. Apart from the delay, bypass paths increase the power consumption of the processor. Thus complete bypassing may have a significant impact in terms of area, cycle time, and power consumption of the processor [6]. Partial bypassing presents a trade-off between the performance, power and cost of a processor. Partial bypassing is therefore an especially valuable technique for application specific embedded processors.

As mentioned earlier, embedded systems are characterized by strict multi-dimensional design constraints, including severe constraints on time-to-market. Short time-to-market makes it imperative to reuse design parts both in hardware and software. Design reuse in compilers, which is one of the most important, time consuming and costly software in an embedded processor system, is facilitated primarily by the means of a retargetable compiler technology. A retargetable compiler, as opposed to a normal compiler, also takes the processor description, as an input parameter. However partial bypassing poses challenges for good quality code generation by retargetable compilers. A good compiler should not only be able to use the bypasses present in the processor, but also avoid the penalty of the bypasses missing in the processor. Although ad-hoc scheduling rules, like "instruction patterns" can be used to generate code for a processor with a given bypass configuration, a more formal and extensible technique is needed for retargetable compilers. The key enabler for this is the ability to accurately detect pipeline hazards. A pipeline hazard detection mechanism is a fundamental capability used in most retargetable scheduling algorithms. Traditional retargetable compilers use the information about the structure of the processor to detect and avoid resource hazards [7], and use constant operation latency of each operation to detect and avoid data hazards [8]. For each operation o, the *operation latency* is defined as a positive integer $ol \in I+$, such that if any data-dependent operation is issued more than ol cycles after issuing o, then there will be no data hazards.

For processors that have no bypassing, or have complete bypassing, the operation latency is a well defined constant. However, for a partially bypassed processor, the operation latency cannot be specified by a constant. In fact the operation latency of an operation depends on the two dependent operations, the dependent operand, the structure of the pipeline and also on the presence/absence of the bypasses. Thus traditional retargetable pipeline hazard detection techniques, that assume a constant operation latency break down in the presence of partial bypassing. There are no existing retargetable pipeline hazard detection techniques for a partially bypassed processor.

In the absence of retargetable pipeline hazard detection mechanisms, it is possible to perform conservative scheduling by using existing techniques. This can be done by using operation latencies obtained by assuming that no bypasses are present. Although conservative scheduling will result in a legitimate schedule even for statically-scheduled processors, it fails to exploit the bypasses present in the processor. The other option is to perform optimistic scheduling using operation latencies obtained by assuming that all bypasses are present. Optimistic scheduling may result in illegitimate schedules for statically scheduled (VLIW) processors, but it is able to use the bypasses present in the processor. However, it incurs penalty due to the missing bypasses in the processor. In fact, it can be shown that pipeline hazard detection using any constant value of operation latency is sub-optimal. Therefore an accurate and retargetable pipeline hazard detection mechanism is needed.

Adding or removing bypasses in a processor is architecture independent (does not affect the instruction set). As a result bypasses in a processor can be changed while still keeping the processor backward compatible. Thus, tuning the bypass configuration is a lucrative option even while developing the next generation of the processor. With incomplete bypassing becoming popular in modern embedded processors, developing retargetable compilers is needed to generate good quality code for them. Bypass-sensitive retargetable code generation therefore not only enables quick and easy adaptation of the compiler to minor changes in the design, but is of paramount importance for rapid and automated design space exploration of processors with partial bypasses.

We solve the problem of retargetable pipeline hazard detection using Operation Tables (OTs). An OT is a mapping between the operands of an operation to the resources and the registers of the processor. An OT captures which processor resources an operation uses in each cycle of it's execution. It can therefore be used to detect resource hazards in a given schedule. An OT also captures the read/write/bypassing of processor registers and can therefore be used to detect data hazards in a given schedule. Thus OTs are able to accurately detect all pipeline hazards in an integrated manner. Most existing scheduling algorithms can leverage our integrated and accurate pipeline hazard detection mechanism to generate better schedules.

2.2 Motivation

Consider the three flavors of bypassing in a simple 5-stage pipeline shown in Figure 2.2, Figure 2.3 and Figure 2.4. In all these pipelines we assume that the write in the register file takes place at the end of the cycle. Thus, if the same register is read and written in a cycle, the old value is read.

Consider the execution of an ADD operation in these pipelines. In absence of any hazards, if the ADD operation is in *F* pipeline stage in cycle i, then it will be in *OR* pipeline stage in cycle $i + 2$. At this time it needs to read the two source registers. The ADD operation will then write back the destination register in cycle $i + 4$, when it reaches the *WB* pipeline stage. The result of the ADD operation can be read from the register file in and after cycle $i + 5$.

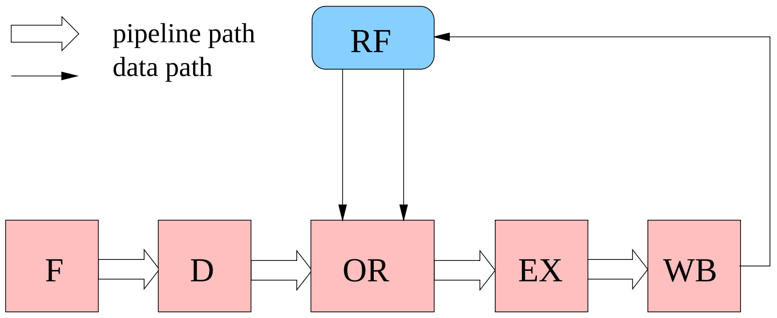


Figure 2.2: A 5-stage processor pipeline with no bypassing

The pipeline in Figure 2.2 does not have any bypasses. There is only one way to read operands, i.e., from *RF*. Thus, the operation latency of ADD is 3 cycles. Any dependent operation should be scheduled at-least 3 cycles after ADD to avoid any data hazard.

Figure 2.3 contains bypasses from both *EX* pipeline stage and *WB* pipeline stage to both the operands of *OR* pipeline stage. This is an example of *complete bypassing.* For completely bypassed pipeline, the operation latency of ADD is 1 cycle. A data dependent

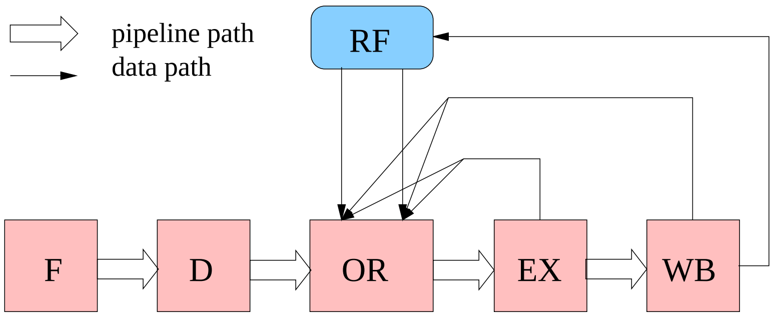


Figure 2.3: A 5-stage processor pipeline with complete bypassing

operation scheduled 1 or 2 cycles after ADD can read the result of ADD from the bypass, while a data dependent operation scheduled 3 or more cycles after ADD can read the result from *RF*. The effect complete bypassing is to reduce the operation latency, resulting in performance improvement. But in either case (in the case of no bypassing, or complete bypassing), *the operation latency can be accurately described by a single value.*

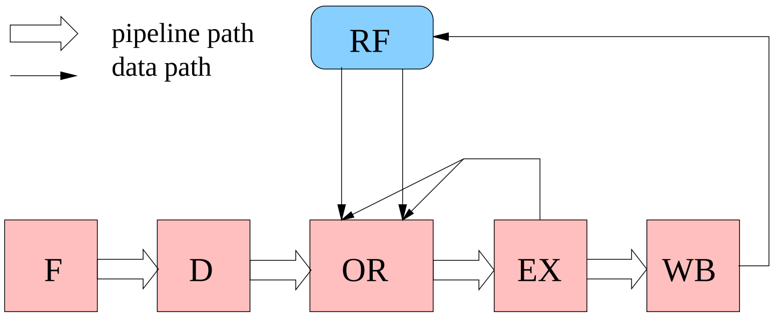


Figure 2.4: A 5-stage processor pipeline with partial bypassing

The pipeline in Figure 2.4 contains bypass only from *EX* pipeline stage to both the operands of *OR* pipeline stage. There is no bypass from *WB* pipeline stage. This is an example of *partial bypassing*. In the pipeline with partial bypassing, scheduling a data dependent operation 1 cycle after ADD will not result in a data hazard, because the result of ADD can be read from *EX* pipeline stage via the bypass. However if the data dependent operation is scheduled 2 cycles after scheduling ADD, there is no way to read the result of ADD. There is a data hazard. But again, if the data dependent operation is scheduled 3 or more cycles after ADD, then the result of the ADD operation can be read from the *RF*. Thus the data hazard can be avoided by scheduling a data dependent operation of ADD 1 cycle or 3 or more cycles after scheduling the ADD operation. The operation latency of ADD in the incompletely bypassed pipeline in Figure 2.2 (c) is denoted by 1, 3, which means that scheduling a data dependent operation 1 or 3 or more cycles after the schedule cycle of ADD will not cause a data hazard, but scheduling the data-dependent operation 2 cycles after the schedule cycle of ADD WILL cause a data hazard.

Thus due to incomplete bypassing, the operation latency of ADD cannot be accurately specified using just one value. Unlike previous approaches that use a single value, in this chapter we show how OTs can be used to accurately pipeline hazards in the presence of such multi-valued operation latencies. The operation latency in the presence of incomplete bypasses is very much linked to the structure of the pipeline and the presence and absence of bypasses, and the path operation takes in the pipeline. Operation Tables define a binding between an operation and the resources it may use and registers it will read/write/bypass in each cycle of it's execution. Using a resource and register model of a processor, OTs can be used to model both the data and resource hazards.

2.3 Related Work

Bypassing was first described in the IBM Stretch [9]. Modern processors heavily use bypasses to avoid data hazards and thereby improve performance. Cohn et. al [10] showed that partial bypassing helps reduce cost with negligible reduction in the performance on the iWarp VLIW processor. Abnous et. al [11], [12] analyzed partial bypassing between VLIW functional units in their 4-integer-unit VIPER processor. They argued that the bypassing cost is minor as compared to the performance benefits achieved in RISC processors, but that complete bypassing is too costly in VLIW processors.

Ahuja et al. [6] discuss the performance and cost trade-off of register bypasses in a RISC-like processor. They manually performed bypass sensitive compilation (operation reordering) on a few benchmarks, and presented results with a relatively coarse cache model. Buss et al. [13] reduce inter-cluster copy operations by placing operand chains into the same cluster and assigning FUs to clusters such that the inter-cluster communication is minimized. The work closest to ours is by Fan et al. [14], in which they describe their bypass customization framework based on bypass sensitive compilation. They focus on VLIW processors and propose an FU-assignment technique to make better use of partial bypasses. However they do not perform instruction reordering. In contrast we propose a bypass-sensitive instruction-reordering technique that is applicable for a wide range of processors.

The concept of Operation Tables (OTs) proposed in this chapter is similar to Reservation Tables (RTs). Reservation Tables (RTs) [15] or Finite State Automata [16], [17] (generated from Reservation Tables) are used to detect resource hazards in retargetable compiler frameworks [18], [4], [3], [7]. RTs model the structure of the processor including the pipeline of the processor and the flow of operations in the pipeline. RTs can thus be used to model resource hazards in a given schedule. However RTs do not model data in the schedule and are thus unable to detect data hazards. Operation Tables, model both the resources and the register information of operations so that *both data and resource hazards* can be effectively captured.

2.4 Processor Model

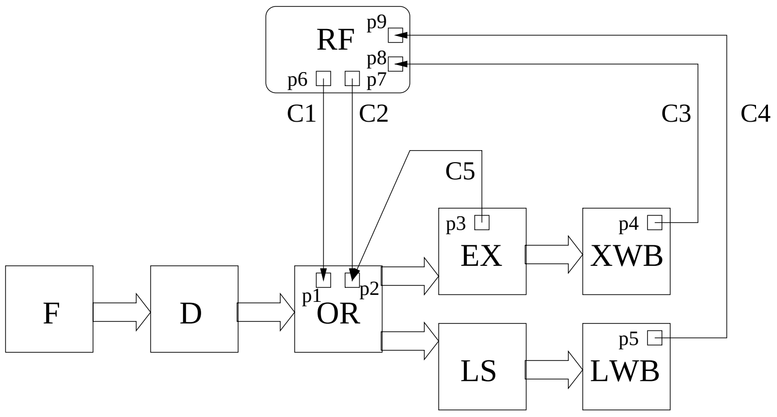


Figure 2.5: Example Pipeline

In this section, we define the processor and operation model. We will then define Operation Table for operations on the processor model depicted in Figure 2.5.

2.4.1 Pipeline Model

A pipelined processor can be divided into pipeline units by the pipeline registers. The processor pipeline can be represented as a Directed Acyclic Graph (DAG) of the pipeline units, $u_i \in U$ which represent the nodes of the DAG, and a directed edge (u_i, u_j) represents that operations may flow from unit u_i to unit u_j. There is a unique "source node", u_0, to which there are no incoming edges. This unit generates operations. Further, some nodes are "sink nodes", which do not have any outgoing edges. These nodes represent writeback units. In the pipeline shown in Figure 2.5, F is the source unit and XWB and LWB are the writeback units. The operations flow along the block arrows.

2.4.2 Operation Model

Each operation $o_i \in O$ supported by the processor is defined using an *opcode* $o_i.opcode$ and a list of source and destination operands, $o_i.sourceOperands$ and $o_i.destOperands$. The *opcode* defines the path of the operation in the processor pipeline. Each source or destination operand, $operand$ is defined by a 3-tuple, <*arg, rf, rn*>, where arg is the argument of the operand, rf is the register file it belongs to, (or IMM for immediate operands), and rn is the register number (or immediate value for immediate operands). The operand argument describes how to read/write the operand. Thus the operation, *ADD R1 R2 5*, has opcode *ADD*, and has one destination operand and two source operands. The destination operand is represented by <*D1, RF, 1*>. The first source operand is represented as <*S1, RF, 2*>, and the third as <*S2, IMM, 5*>.

2.4.3 Pipeline Path of Operation

The pipeline path of an operation o_i is the ordered list of units that an operation flows through, starting from the unique source unit u_0, to at least one of the writeback units. Each unit $u_i \in U$ contains a list of operations that it supports, $u_i.opcodes$. The add operation, *ADD R1 R2 5* has opcode *ADD*, and the pipeline units F, D, OR, EX and XWB have the *ADD* operation in the list of opcodes they support.

2.4.4 Register File

We define a register file as a group of registers that share the read/write circuitry. A processor may have multiple register files. The processor in Figure 2.5 has a register file named *RF*.

2.4.5 Ports in Register File

A register file contains read ports and write ports to enable reading and writing of registers from and to the register file. Register operands can be read from a register file

rf via read ports, $rf.readPorts$, and can be written in rf via write ports, $rf.writePorts$. Register operands can be transferred via ports through register connections. The register file *RF* in the processor in Figure 2.5, has two read ports ($p6$ and $p7$) and two write ports ($p8$ and $p9$).

2.4.6 Ports in Pipeline Units

A pipeline unit, u_i can read register source operands via its read ports, $u_i.readPorts$, write result operands via its write ports, $u_i.writePorts$, and bypass results via its bypass ports, $u_i.bypassPorts$. Each port in a unit is associated with an argument arg, which defines the operands that it can transfer. For example a *readPort* of a unit with argument *S1* can only read operands of argument *S1*. In the processor in Figure 2.5, pipeline unit *OR* has 2 read ports, $p1$ and $p2$ with arguments, *S1* and *S2* respectively. The units, *XWB* and *LWB* have write ports $p4$ and $p5$ respectively with arguments *D1*, and *D2* respectively while *EX* has a bypass port $p3$ with argument $D1$.

2.4.7 Register Connection

A register connection *rc* facilitates register transfer from a source port *rc.srcPort* to destination port *rc.destPort*. In the processor diagram in Figure 2.5, the pipeline unit *OR* can read two register source operands, first from the register file *RF* (via connection *C1*), and second from *RF* (via connection *C2*) as well as from *EX* (via connection *C5*). The register connection C5 denotes a bypass.

2.4.8 Register Transfer Path

Register transfers can happen from a register file to a unit (register read), from a unit to a register file (a writeback operation), and even between units (register bypass). The register transfers in our processor are modeled explicitly via ports. A register transfer path is the list of all the resources used in a register transfer, i.e., the source port, the register connection, the destination port, and the destination register file or unit.

Operation Table Definition		
OperationTable	:=	{ otCycle }
otCycle	:=	unit ros wos bos dos
ros	:=	ReadOperands { operand }
wos	:=	WriteOperands { operand }
bos	:=	BypassOperands { operand }
dos	:=	DestOperands { regNo }
operand	:=	regNo { path }
path	:=	port regConn port regFile

Table 2.1: Operation Table Definition

2.5 Operation Table

An Operation Table (OT) describes the execution of an operation in the processor. Table 2.1 describes the grammar and structure OTs. An OT is a DAG of *OTCycles*, each *OTcycle* describes what happens in each execution cycle, while the directed edges between *OTCycles* represent the time-order of *OTCycles*. Each OTCycle describes the unit in which the operation is, and the operands it is reading *ros*, writing *wos* and bypassing *bos* in the execution cycle. The destination operands *dos* are used to indicate the destination registers, and are required to model the dynamic scheduling algorithms in the processor. Each *operand* that is transferred (i.e., read, written, or bypassed) is defined in terms of the register number, *regNo*, and all the possible *paths* to transfer it. A *path* is descibed in terms of the ports, register connections and the register file involved in the transfer of the operand.

Table 2.2 shows the OT of the add operation, *ADD R1 R2 R3* on the partially bypassed pipeline shown in Figure 2.5. In the absence of any hazards, the add operation executes in *5* cycles, therefore the OT of the add operation contains *5 otCycles*. In the first cycle of its execution, the add operation needs the *F* pipeline stage, and in the second cycle it needs *D* pipeline stage. In the third cycle, the add operation occupies *OR* pipeline stage and needs to read its source operands *R2* and *R3*. All the paths to read each *readOperand* are listed. The first *readOperand, R2* can be read only from the *RF* via connection *C1*. There are two possible paths to read the second operand. First is from RF via ports p7 and p2 and connection C2. The second path is from Ex via ports p3 and p2 and connection C5. Since the sources are read in this cycle, the *destOperands* are listed. In the fourth cycle the add operation is executed and needs *EX* pipeline stage. The result of the operation

	Operation Table of ADD R1 R2 R3			
1	F			
2	D			
3	OR			
		ReadOperands		
			R2	
				p1, C1, p6, RF
			R3	
				p2, C2, p7, RF
				p2, C5, p3, EX
		DestOperands		
			R1, RF	
4	EX			
		BypassOperands		
			R1	
				p3, C5, p2, OR
5	WB			
		WriteOperands		
			R1	
				p4, C3, p8, RF

Table 2.2: Operation Table of ADD R1 R2 R3

R1 is bypassed via connection *C5*. It can be read as the second operand of the operation occupying the *OR* unit. *WB* pipeline stage is needed in the fifth cycle. In the otCycle the result of the add operation *R1* is written back to *RF* via connection *C3*.

2.6 Bypass Register File

The OT of an operation lists all the paths to read each operand. In the presence of bypasses, there may be multiple ways to read an operand. For example, in the Intel XScale processor, an operand can be read from the register file and 7 bypasses. Thus there can be 8 paths to read an operand. Listing down all the paths to read the operands makes the OT description not only long, but also error-prone. To reduce the complexity of specification, the concept of Bypass Register File (BRF) is used.

2.6.1 Bypass Register File

A Bypass Register File (BRF) is a virtual register file for each operand that is read in the processor pipeline. All the bypasses that are attached to the operand write to the

BRF, and the operand can be read from the normal register files or the BRF. The semantics of BRF differ from a regular register file in the sense that the register values are valid for only one cycle. Only one BRF is needed for each operand that can accept values from the bypasses. This greatly reduces the complexity and redundency in the specification of OTs.

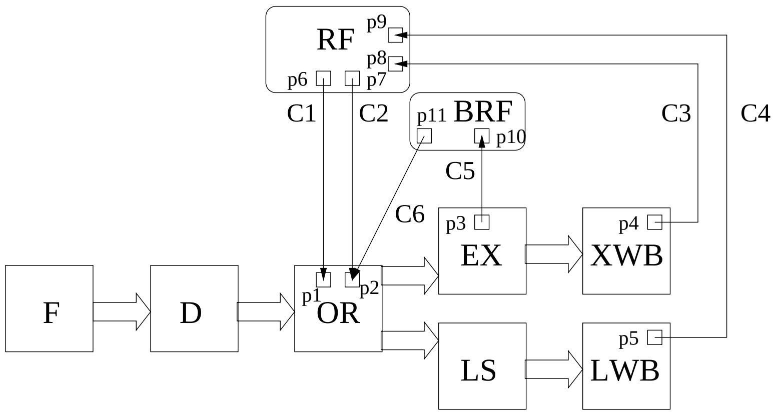


Figure 2.6: Example Pipeline with BRF

Figure 2.6 shows the processor pipeline model with Bypass Register File abstraction. All the bypasses to port $p2$ of unit OR write into the BRF, and the second operand that has to be read via port $p2$, can now be read either from the register file RF via connection $C2$, or from BRF via connection $C6$.

Table 2.3 shows the OT of the operation $ADD\ R1\ R2\ R3$ using the BRF abstraction. The only difference is that the EX unit now bypasses the result to the BRF though ports $p3$, and $p10$, and connection $C5$, and that the second operand in OR is read via the register file RF or via the BRF.

Therefore using the BRF abstraction, only one Bypass Register File is needed for each machine operand that can be read and it accepts bypasses.

2.7 Hazard Detection using Operation Tables

Operation Tables can be used to detect pipeline hazards in a processor, for a given schedule of instructions. Hazard detection using OTs requires that the state of the machine (processor) be maintained to reflect the current schedule. The state of the machine is defined in Table 2.4.

A *machineState* is a ordered list (square brackets) of *macCycle*. Each *macCycle*

	Operation Table of ADD R1 R2 R3			
1	F			
2	D			
3	OR			
		ReadOperands		
			R2	
				p1, C1, p6, RF
			R3	
				p2, C2, p7, RF
				p2, C6, p11, BRF
		DestOperands		
			R1, RF	
4	EX			
		BypassOperands		
			R1	
				p3, C5, p10, BRF
5	WB			
		WriteOperands		
			R1	
				p4, C3, p8, RF

Table 2.3: Operation Table of ADD R1 R2 R3 with BRF

Machine State		
machineState	:=	[macCycle]
macCycle	:=	Resources, RF, BRF

Table 2.4: Machine State

denotes the state of the machine resources (whether they are busy or free), and the registers in the register files (whether they are available for read, or not). The state of the *Bypass Register File* also a part of the *macCycle*.

The function *DetectHazard* described in Figure 2.7 is the main pipeline hazard detection function. It detects both the data and resource hazards if the operation *op* is scheduled at time *t* in a given *machineState*. The function *DetectHazard* tries to schedule each *otCycle* of the operation in the machine states starting from time *t*. It reports a hazard if there is a hazard in scheduling any *otCycle* in the corresponding *macCycle*.

The function *DetectCycleHazard*, described in Figure 2.8 detects a hazard when an *otCycle* is scheduled in a *macCycle*. The function reports a hazard if a resource that is required in the *otCycle* is not present in the *macCycle*. A hazard is reported if any *readOperand* cannot be read (lines 04-07), or *writeOperand* cannot be written (lines 08-12). To avoid WAW (Write After Write), a hazard is reported when a *destOperand* is not

```
bool DetectHazard(machineState, Operation op, Time t)
01: for $(i = 0; i < op.OT.length; i++)$
02:   if $(DetectCycleHazard(machineState[t + i], op.OT[i]))$
03:     return TRUE
04:   endIf
05: endFor
06: return FALSE
```

Figure 2.7: Detect Hazard when an operation is scheduled

present in the Register File (lines 13-18).

The function *AvailRP* in Figure 2.9 tells whether the register *reg* can be read via any *paths* in the *macCycle*. A register can be read by a *path* if the register is present in the *RegFile* (line 04) and all the resources required to read the register from the *RegFile* are available (line 03). A register can be read if it can be read by any *path* (line 01, 05). Similarly, the function *AvailWP* indicates if the register *reg* can be written via any of the *paths* in the *macCycle*. The register can be written in the *RegFile* in cycle *macCycle* (line 13) if all the resources in any path (line 10) are available in the *macCycle* (line 12).

The function *DetectHazard* is thus able to tell, if there is a hazard in scheduling an operation in certain cycle, given the state of the machine. For this to function correctly, the state of the machine needs to be maintained. The function *AddOperation* in Figure 2.10 updates the machine state after scheduling operation *op* in cycle *t*. It finds the earliest *macCycle* to schedule each *otCycle* without a hazard (lines 03-04), and then updates the *macCycle*. Each *machineState* is updated by scheduling an *otCycle* by the function *AddCycle* (line 06).

The function *AddCycle* in Figure 2.11 updates a *macCycle* by scheduling an *otCycle* in it. A *macCycle* is updated by removing all the *Resources* required in *otCycle* from the *Resources* in *macCycle* (line 01). All the required resources for the operand reads (lines 02-05) and writes (lines 06-11) are also marked as busy. If there are *DestOperands*, *RemRegFromRegFile* removes the *Register* from *RF* in the later cycles (lines 12-15). The function *AddRegToRegFile*, adds the *WriteOperands* to *RF* in the later cycles (line 10).

In the next sections we show that the *DetectHazard* function, and the *AddOperation* function can be used to detect all the pipeline hazards using Operation Tables. Thereafter we discuss how these fundamental functions can be used to improve most existing standard

```
bool DetectCycleHazard(macCycle, otCycle)
/* resource hazard */
01: if ($otCycle.Resources \not\subset macCycle.Resources$)
02:   return TRUE

/* all sources can be read*/
03: foreach ($ro \in otcycle.ReadOperands$)
04:   if ($AvailRP(ro.Register, ro.Paths, macCycle) == \phi$))
05:     return TRUE
06:   endIf
07: endFor

/* all dests can be written */
08: foreach ($wo \in otcycle.WriteOperands$)
09:   if ($AvailWP(wo.Register, wo.Paths, macCycle) == \phi$))
10:     return TRUE
11:   endIf
12: endFor

/* dest is not available */
13: foreach ($do \in otcycle.DestOperands$)
14:   $regFile = do.RegisterFile$
15:   if ($do.Register \notin macCycle.regFile$)
16:     return TRUE
17:   endIf
18: endFor

/* no hazard */
19: return FALSE
```

Figure 2.8: Detecting Hazards when a cycle of Operation Table is scheduled

scheduling algorithms.

2.8 Illustrative Example

Consider scheduling the sequence of three operations in the pipeline in Figure 2.6.

MUL R1 R2 R3 ($R1 \leftarrow R2 \times R3$)

ADD R4 R2 R3 ($R4 \leftarrow R2 + R3$)

SUB R5 R4 R2 ($R5 \leftarrow R4 - R2$)

The OTs of ADD and SUB are similar, except for the register indices. The MUL

```
path AvailRP(reg, paths, macCycle)
01: foreach ($path \in paths$)
02:   $regFile = path.RegisterFile$
03:   if ($path.Resources \subset macCycle.Resources$)
04:     if ($reg \in macCycle.regFile$)
05:       return $path$
06:     endIf
07:   endIf
08: endFor
09: return $\phi$

path AvailWP(reg, paths, macCycle)
10: foreach ($path \in paths$)
11:   $regFile = path.RegisterFile$
12:   if ($path.Resources \subset macCycle.Resources$)
13:     return $path$
14:   endIf
15: endFor
16: return $\phi$
```

Figure 2.9: Finding the available read and write path

operation uses the same resources but spends two cycles in the *EX* pipeline stage. An operation bypasses the results only after the execution has finished. Thus a valid bypass from *EX* pipeline stage will be generated only in the second cycle of execution of MUL.

Since MUL occupies the *EX* pipeline stage for two cycles, a resource hazard should be detected between the MUL and ADD operation. SUB requires the result of ADD operation as the first operand, for which there is no bypass, so there should be a data hazard. We illustrate the detection of hazards by scheduling these three operations in-order. Initially We assume that all the resources are free and that all the registers are available in the *RF*. There is no hazard when MUL is scheduled in the first cycle. Figure 2.12 shows the *machineState* after MUL is scheduled by *AddOperation*.

If we try to schedule ADD in the next cycle, *DetectHazard* detects a resource hazard. There is a resource hazard when the fourth *otCycle* of ADD is tried in the fifth *macCycle*. The resource *EX* is not free in the *macCycle*. Figure 2.13 shows the *machineState* after scheduling ADD in the second cycle using *AddOperation*.

Now in the existing schedule in Figure 2.13, if we try to schedule SUB in the third cycle, there is a data conflict. The third *otCycle* of SUB cannot read R4 from *RF*. *AvailRP*

```
void AddOperation(machineState, op, t)
01: $j = t$
02: for ($i = 0; i < op.OT.length; i++$)
03:   while ($DetectHazard(machineState[j], op.OT[i])$)
04:     $j++$
05:   endWhile
06:   $AddCycle(machineState[j], op.OT[i])$
07: endFor
```

Figure 2.10: Update the state of the machine

returns ϕ because even though the connection *C1* is free, R4 is not present in *RF*. The data hazard is resolved in the eighth cycle of *machineState*. Figure 2.14 shows *machineState* after SUB is scheduled in the third cycle using *AddOperation*.

Thus Operation Tables can be used to accurately detect both data and resource conflicts, even in the presence of incomplete bypassing.

2.9 Integrating OTs in a Scheduler

Detection of pipeline hazards is a fundamental problem in scheduling. Our OT-based approach generates accurate hazard detection information, allowing any traditional scheduling algorithm to perform better. For the sake of illustration, we demonstrate how to modify a simple list scheduling algorithm to use Operation Tables. We believe OTs can similarly be integrated into other scheduling formulations. The scheduling problem is to schedule the vertices of a data dependence graph $G = (V, E)$, where each vertex corresponds to an operation, and there is an edge between v_i and v_j if v_j uses the result of v_i. Vertex v_0 and v_n are the unique start and end vertices. The function $parents(v)$ gives a list of all the parents of v.

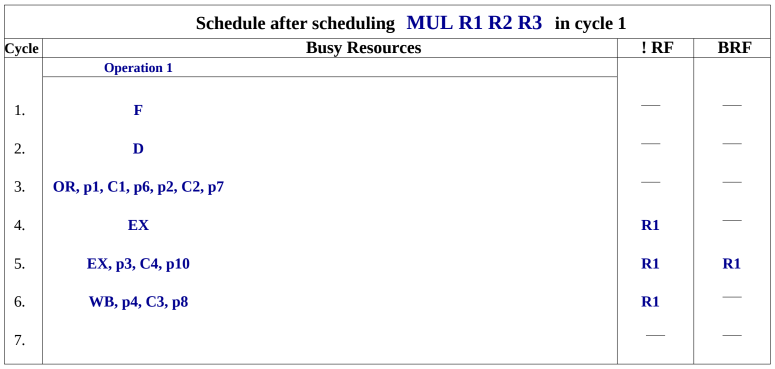
Figure 2.15 maintains three sets of vertices, U, the unscheduled vertices, F, the frontier vertices or the vertices that are ready to be scheduled, and S, the vertices that have been scheduled (line 01). We initialize the algorithm by scheduling the vertex start vertex v_0. Therefore, $U = V - v_0$, $F = \phi$, and $S = v_0$. The schedule time for each vertex is initialized to 0 ($schedTime[v] = 0$, $\forall v \in V$) (lines 02-04). For scheduling, the frontier set of unscheduled vertices is computed in each step. All unscheduled vertices ($v \in U$) whose parents have been scheduled ($parents(v) \subset S$) belong to the frontier set. (line 06) The vertices in the

```
void AddCycle(macCycle, opcycle)
/* make the resources busy */
01: $macCycle.Resources -= macCycle.Resources$

/* mark the resources used in read as busy */
02: foreach ($ro \in otcycle.ReadOperands$)
03:   $path = AvailRP(ro.Register, ro.Paths, macCycle)$
04:   $macCycle.Resources -= path.Resources$
05: endFor

/* mark the resources used in write as busy */
06: foreach ($wo \in otcycle.WriteOperands$)
07:   $reg = wo.Register$
08:   $path = AvailWP(wo.Register, wo.Paths, macCycle)$
09:   $macCycle.Resources -= path.Resources$
10:   $AddRegToRegFile(reg, path.RegisterFile, j)$
11: endFor

/* remove the dest register */
12: foreach ($do \in otcycle.DestOperands$)
13:   $reg = do.Register$
14:   $regFile = do.RegisterFile$
15:   $RemRegToRegFile(reg, regFile, j)$
16: endFor
```

Figure 2.11: Update a machine cycle

frontier set are sorted by some priority function (line 07), and the vertex with the least cost is picked for scheduling (line 08). The minimum schedule time for v is the maximum of the schedule time plus the latency of each parent vertex p (line 09). The vertex v is then scheduled in the first cycle, when it does not cause a hazard (lines 10-13). Different implementations of list scheduling mainly differ in the priority function for the frontier set.

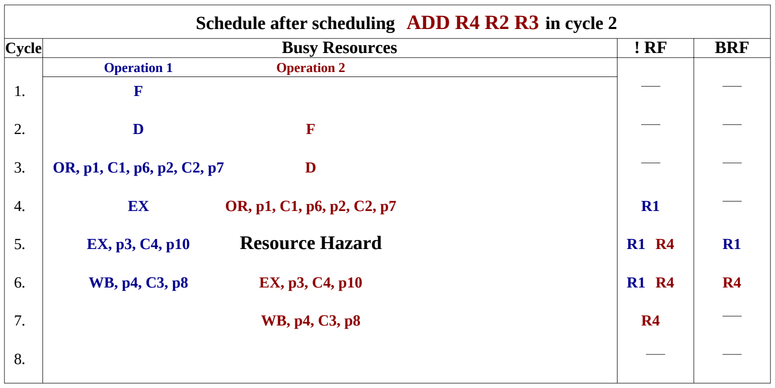
However in the presence of partial bypassing, the operation latency of an operation is not sufficient to avoid all the data hazards; Operation Tables are needed. The traditional list scheduling algorithm can be very easily modified to make use of the *DetectHazard* and *AddOperation* functions to schedule using Operation Tables. Figure 2.16 shows the same list scheduling algorithm that uses Operation Tables for pipeline hazard detection. The only modification required is to change the *DetectHazard* function (line 09) and the *AddOperation* function (line 12). Our new Operation Table based *DetectHazard* function defined in Figure 2.7 is used. The *AddOperation* function is needed to update the



Schedule after scheduling **MUL R1 R2 R3** in cycle 1

Cycle	Busy Resources	! RF	BRF
	Operation 1		
1.	F	—	—
2.	D	—	—
3.	OR, p1, C1, p6, p2, C2, p7	—	—
4.	EX	R1	—
5.	EX, p3, C4, p10	R1	R1
6.	WB, p4, C3, p8	R1	—
7.		—	—

Figure 2.12: Schedule after scheduling MUL R1 R2 R3



Schedule after scheduling **ADD R4 R2 R3** in cycle 2

Cycle	Busy Resources		! RF	BRF
	Operation 1	Operation 2		
1.	F		—	—
2.	D	F	—	—
3.	OR, p1, C1, p6, p2, C2, p7	D	—	—
4.	EX	OR, p1, C1, p6, p2, C2, p7	R1	—
5.	EX, p3, C4, p10	Resource Hazard	R1 R4	R1
6.	WB, p4, C3, p8	EX, p3, C4, p10	R1 R4	R4
7.		WB, p4, C3, p8	R4	—
8.			—	—

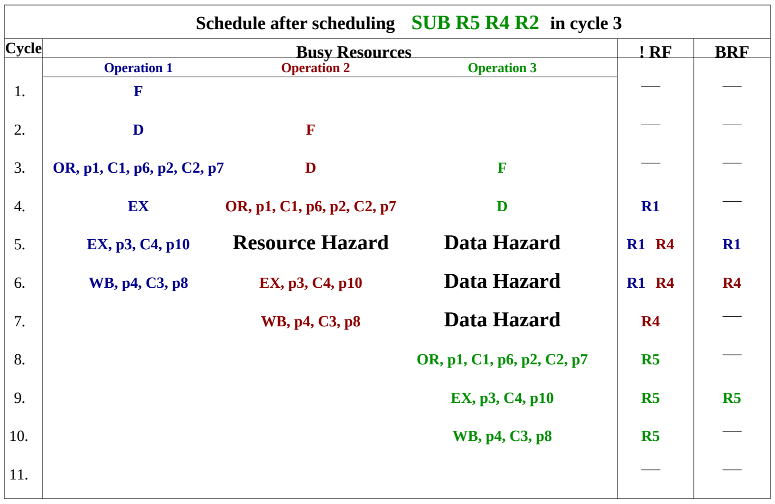
Figure 2.13: Schedule after scheduling ADD R4 R4 R3

machineState that is a input parameter in the *DetectHazard* function.

Thus we have demonstrated the integration of the Operation Table based pipeline hazard detection mechanism into the standard list scheduling algorithm. Pipeline hazard detection being a very important and distinct part in most scheduling algorithms enables easy swapping by our more accurate OT-based technique. Thus most existing scheduling algorithms should be able to leverage the accurate pipeline hazard detection mechanism to generate better schedules.

2.10 Effectiveness of OT-based Bypass-sensitive Compiler

To demonstrate the need and efficacy of Operation Tables, we perform experiments on the popular embedded processor, the Intel XScale [19], employed in wireless and hand-held devices. The Intel XScale provides high code density, high performance and low power, all at the same time. Figure 2.17 shows the 7-stage out-of-order superpipeline of XScale.



Schedule after scheduling SUB R5 R4 R2 in cycle 3

Cycle	Busy Resources			! RF	BRF
	Operation 1	Operation 2	Operation 3		
1.	F			—	—
2.	D	F		—	—
3.	OR, p1, C1, p6, p2, C2, p7	D	F	—	—
4.	EX	OR, p1, C1, p6, p2, C2, p7	D	R1	—
5.	EX, p3, C4, p10	Resource Hazard	Data Hazard	R1 R4	R1
6.	WB, p4, C3, p8	EX, p3, C4, p10	Data Hazard	R1 R4	R4
7.		WB, p4, C3, p8	Data Hazard	R4	—
8.			OR, p1, C1, p6, p2, C2, p7	R5	—
9.			EX, p3, C4, p10	R5	R5
10.			WB, p4, C3, p8	R5	—
11.				—	—

Figure 2.14: Schedule after scheduling SUB R5 R4 R2

XScale implements dynamic scheduling using register scoreboarding, and has a partially bypassed pipeline. We present experimental results on the benchmarks from MiBench [20] suite, which are the representative of typical embedded applications. To estimate the performance of compiled code we have developed a cycle-accurate simulator of the Intel XScale processor pipeline. Our simulator structurally models the Intel XScale pipeline and its performance measurements have been validated against the 80200 evaluation board [21]. The performance measurement of our cycle-accurate simulator is accurate to within 7% of the evaluation board. This accuracy is good enough to perform processor pipeline experiments with reasonable fidelity.

Figure 2.18 shows the experimental setup for these experiments. We first compile the applications using GCC cross compiler for XScale. The benchmarks were compiled using the -O3 option to optimize for performance. We then simulate the compiled code on our XScale cycle accurate simulator and measure the number of execution cycles ($gccCycles$).

We read the executable and generate the control flow graph and other data dependency data structures. We perform OT-based scheduling on each basic block of the program, and generate the executable again. Our within-basic block scheduling algorithm is very simple. We enumerate all the possible schedules and consider only the first 1000 schedules. The instructions are re-ordered to match with the best performing schedule and the new executable is generated. We simulate the new executable on the same XScale cycle accurate simulator and measure the number of execution cycles ($otCycles$). The percentage

```
ListSchedule(V)
01: $U = V - v_0; F = \phi; S = v_0$

/* initialize */
02: foreach $(v \in V)$
03:   $schedTime[v] = 0$
04: endFor

/* list schedule */
05: while $(U \neq \phi)$
06:   $F = \{v | v \in U, parents(v) \subset S\}$
07:   $F.sort()$ /* some priority function */
08:   $v = F.pop()$
09:   $t = MAX(schedTime(p) + p.OL),\ p \in parents(v)$
10:   while $(DetectHazard(v, t))$
11:     $t{+}{+}$
12:   endWhile
13:   $schedTime[v] = t$
14: endWhile
```

Figure 2.15: Original List Scheduling Algorithm

performance improvement is computed as $\frac{gccCycles - otCycles}{gccCycles} \times 100$.

We perform exhaustive scheduling within the basic blocks. Exhaustive scheduling must have exponential time complexity, but data dependencies limit the number of possible schedules. As a result the OT-based rescheduling time for all the benchmarks is below 1 minute.

Figure 2.19 plots the percentage performance improvement over various benchmarks. The graph shows that our code generation scheme can generate up to 20% better performing code than the best performing code generated by GCC. *susan.corners* is an image processing algorithm that finds the corners in a given image. Our compiler was able to detect data conflicts in two innermost loops of the corner detection algorithm, and was able to find a schedule that avoided the conflict. In the *bitcount* benchmark, the scheduling could not find a better schedule than GCC in the frequently executed loops. It could find at least one instance of data hazard (undetected by GCC) in some other loop, and was able to avoid it, but since the loop was not among the most frequently executed loop the performance difference was not significant. In the *qsort* benchmark, although our detection hazard could detect some sub-optimal schedules, it was not possible to avoid them by re-ordering alone.

```
ListScheduleUsingOTs(V)
01: $U = V - v_0; F = \phi; S = v_0$

/* initialize */
02: foreach ($v \in V$)
03:   $schedTime[v] = 0$
04: endFor

/* list schedule */
05: while ($U \neq \phi$)
05:   $F = \{v | v \in U, parents(v) \subset S\}$
06:   $F.sort()$ /* some priority function */
07:   $v = F.pop()$
08:   $t = MAX(schedTime(p)),\ p \in parents(v)$
09:   while ($DetectHazard(machineState, v.OT, t)$)
10:     $t{+}{+}$
11:   endWhile
12:   $AddOperation(machineState, v.OT, t)$
13:   $schedTime[v] = t$
14: endWhile
```

Figure 2.16: List Scheduling algorithm using Operation Tables

We surmise that if OT-based hazard detection technique is implemented as a first-class technique in the main compiler flow, much better results can be achieved. Another reason that reduces the effectiveness of our schedule is due to the variation of latencies of operations, especially the memory latencies. OT-based compiler generates a precise schedule for a given flow of instructions in the pipeline and their latencies. Any variation in the latencies disturbs the quality of the schedule generated. In the benchmark *susan.edges*, the high rate of cache misses disturbed the generated schedule so much that very little improvement was achieved. Although the benefits achieved by a scheduler using OTs may be small, *it is always beneficial*. In fact for our set of benchmarks, OT-based compiler on an average generates 8% better performing schedule than the best performing schedule generated by GCC.

2.11 Compiler-in-the-Loop Partial Bypass Exploration

Traditionally the decision of which bypasses to add/remove is based on the designer's intuition and/or simulation-only exploration. The traditional method of exploring partial

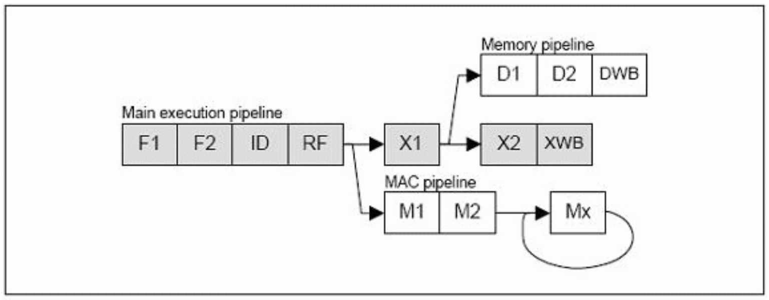


Figure 2.17: 7-stage pipeline of XScale

bypasses i.e. *simulation-only* exploration is performed by measuring the performance of the same compiled code (binary) on processor models with different bypass configurations. The configuration with the best performance is chosen. However, once a bypass configuration is chosen, a "production compiler" is developed for the chosen bypass configuration. Although it takes a lot of time and effort to develop the production compiler, finally it is able to exploit the bypasses present in the processor. It has been shown that tuning the compiler for the bypass configuration has significant impact on the performance of a partially bypassed processor [22]. This implies that the performance estimation done by the simulation-only exploration incurs significant errors. Furthermore in a simulation-only exploration, since the code that executes on the processor may not be the correct representative of the code that will be finally executed on the processor, it leads to inaccuracies in other estimates e.g. power. There is thus a crucial need of a bypass-sensitive compiler-in-the-loop exploration of partial bypassing in embedded processors.

Embedded systems, which are characterized by multi-dimensional design constraints including power, performance and cost, critically require an exploration framework which is able to accurately evaluate the performance, area and energy of each design alternative and thus perform meaningful multi-dimensional trade-offs.

To address these issues, we developed PBExplore: A Compiler-in-the-Loop Framework to explore Partial Bypassing in processors. PBExplore evaluates the performance of a bypass configuration by generating code for the processor with given bypass configuration and simulates the generated code on cycle accurate simulator of the processor with the same bypass configuration. PBExplore also synthesizes the bypass control logic and evaluates the area and energy overhead of the bypass configuration. Thus PBExplore is able to effectively perform meaningful multi-dimensional (performance-area-power) trade-offs among bypass

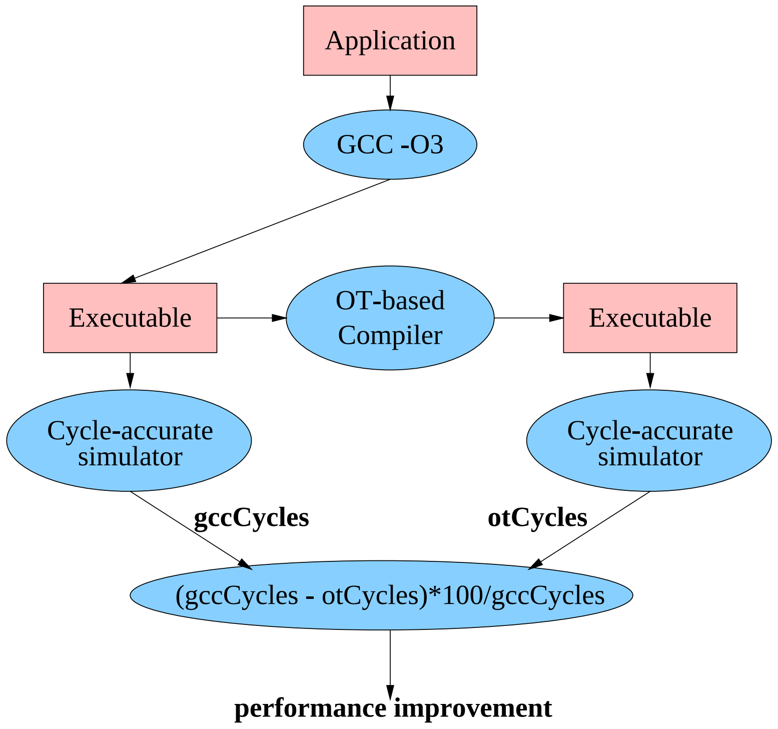


Figure 2.18: Experimental Setup

configurations. This makes PBExplore a valuable assist for designers of programmable embedded systems.

2.11.1 PBExplore: A Compiler-in-the-Loop Exploration Framework

PBExplore is driven by bypass configuration as shown in Figure 2.20. All the bypasses present in the processor are described in the bypass configuration. A bypass is defined in terms of the pipeline stage where it is generated, and the operand that can use it. The application is compiled using a bypass-sensitive compiler that is parameterized on the bypass configuration. The generated executable is then simulated on a cycle accurate simulator that is parameterized on the same bypass configuration. The cycle accurate simulator reports the runtime (in cycles) for the application. The bypass configuration is used to synthesize the bypass control logic and estimate the area overhead of bypasses. A Power simulator uses the synthesized bypass control logic, and the input stimuli in each cycle (generated by the cycle accurate simulator) to estimate the energy consumed by the bypass control logic for the execution of the application. Thus PBExplore is able to make an accurate estimation of performance (cycles of execution), area and energy consumption overhead for each bypass

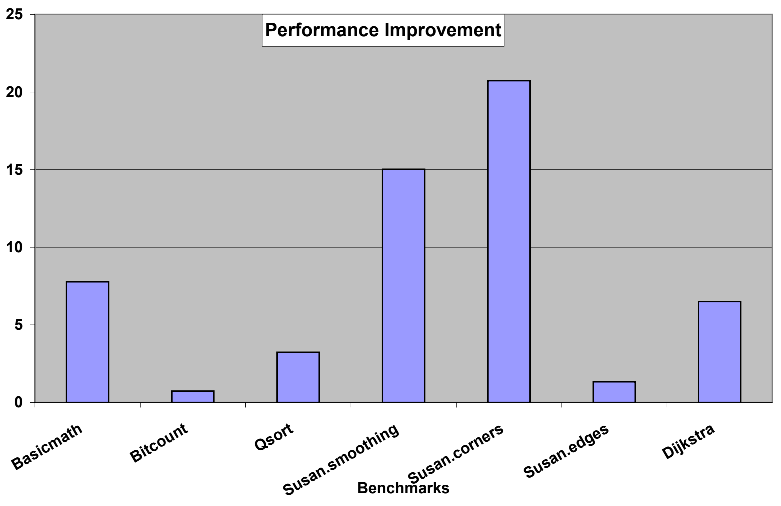


Figure 2.19: Experimental Results

configuration. We now describe the different components of PBExplore.

2.11.2 Area and Energy Overhead Estimation

We quantify the area and energy consumption overhead of bypassing by synthesizing the bypass control logic for each bypass configuration. Figure 2.21 shows the bypass logic for the second operand in the OR pipeline stage in the pipeline in Figure 2.21, which receives only one bypass (from the EX pipeline stage). Each operand can potentially receive bypass from each pipeline stage. Of course for real processors that have large number of such bypasses for each operand, the bypass control logic scales and result in significant area and energy consumption overhead. Each pipeline stage that is a source of a bypass, generates a bypass value, a bypass valid and a bypass register number. If the operand to be read matches any of the incoming bypass register numbers, then the corresponding bypass value is chosen, otherwise the value from the register file is chosen. We synthesize the bypass logic using the Synopsys Design Compiler[23] and estimate the area overhead of the bypass control logic. Synopsys Power Estimator[23] is then used to simulate this bypass control logic with the input stimuli generated by the cycle accurate simulator to estimate the energy consumption of the bypass control logic.

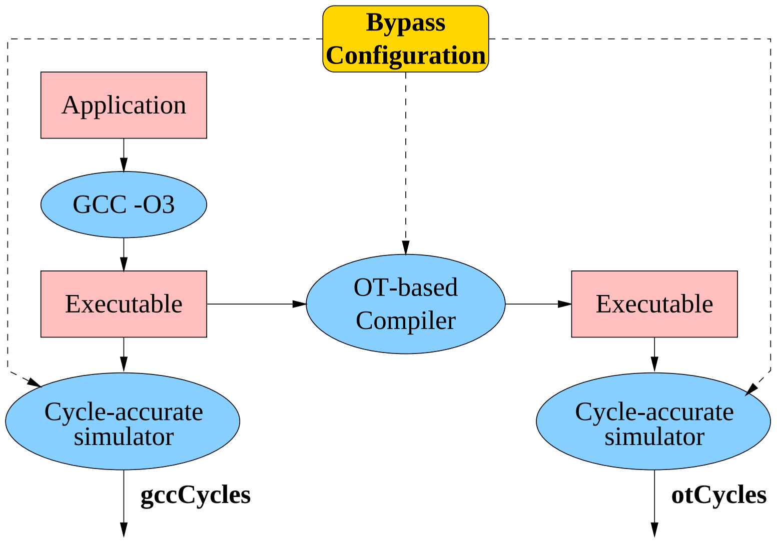


Figure 2.20: PBExplore: A Compiler-in-the-Loop Framework for Partial Bypass Exploration

2.11.3 Simulation-only Exploration

To demonstrate the need, usefulness and capabilities of PBExplore, we perform several experiments on the Intel XScale[19] architecture. XScale has three execution pipelines, the X pipeline (units X1, X2, and XWB), the D pipeline (units D1, D2 and DWB), and M pipeline (units M1, M2 and Mx(referred to as MWB)). For our experiments we assume that 7 pipeline stages, X1, X2, XWB, M2, MWB, D2 and DWB can bypass to all the 3 operands in RF. Thus there are $7 \times 3 = 21$ different bypasses in XScale. No computation finishes before or in the pipeline units M1 and D1, thus there are no bypass connection from these units.

In the XScale pipeline model, we vary whether a pipeline stage bypasses its result or not. If a pipeline stage bypasses, all the operands can read the result. Thus there are $2^7 = 128$ possible bypass configurations. Figure 2.22 plots the runtime (in execution cycles) of the *bitcount* benchmark for all these configurations using simulation-only (dark diamonds), and compiler-in-the-loop (light squares) exploration. We make two important observations from this graph. The first is that *all* the light squares are below their corresponding dark diamonds, indicating that the execution cycles evaluated by compiler-in-the-loop exploration is less than the execution cycles evaluated by the simulation-only exploration. This implies that the bypass-sensitive compiler is able to effectively take advantage of the bypass configuration, and is generating good quality code for *each* bypass

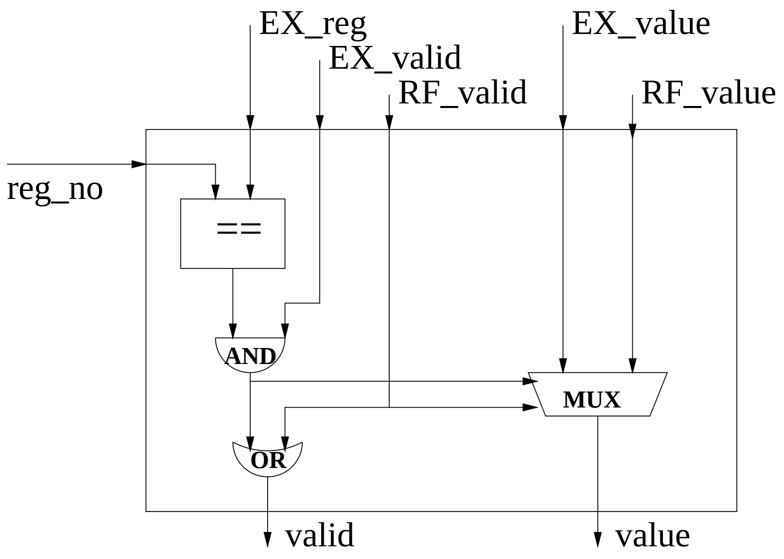


Figure 2.21: Bypass control logic of second operand

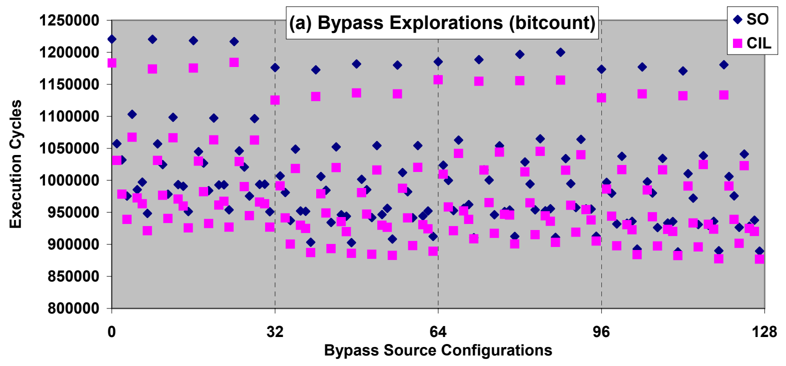


Figure 2.22: Simulation-only vs. Compiler-in-the-Loop Exploration

configuration. The second observation is that the difference in the execution cycles for a bypass configuration can be up to 10%, implying that the performance evaluation by simulation-only exploration can be up to 10% inaccurate.

A case can be made for simulation-only exploration by arguing that the error in exploration is important only if it leads to a difference in trend. To counter this claim we will now zoom into this graph and show that simulation-only exploration and compiler-in-the-Loop exploration result in different trends, and may lead to different design decisions. Figure 2.23 is a zoom-in of Figure 2.22 and shows the explorations when only the X-bypasses are varied, while the rest are present. To bring out the difference in trends, the bypass configurations in this graph are sorted in the order of execution cycles as evaluated by simulation-only

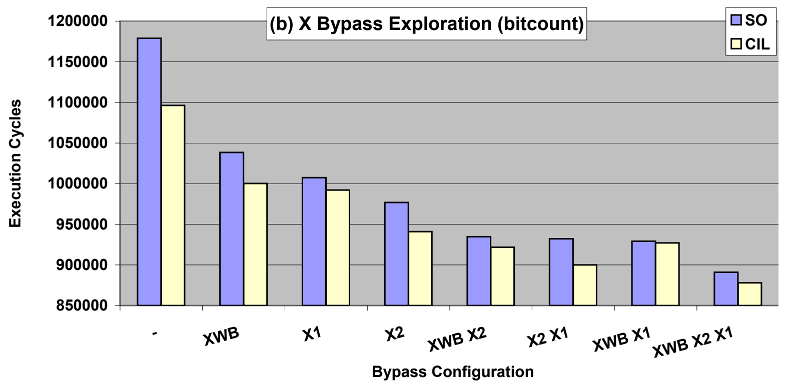


Figure 2.23: X-bypass Exploration for bitcount benchmark

exploration. Figure 2.23 shows that as per the simulation-only approach, all configurations with bypasses from two stages in the X-pipeline are similar, i.e. the execution cycles for configurations $< X2\ X1 >$, $< XWB\ X1 >$ and $< XWB\ X2 >$ are similar. However, our bypass-sensitive compiler is able to exploit the configuration $< X2\ X1 >$ better than other configurations with two X-bypasses.

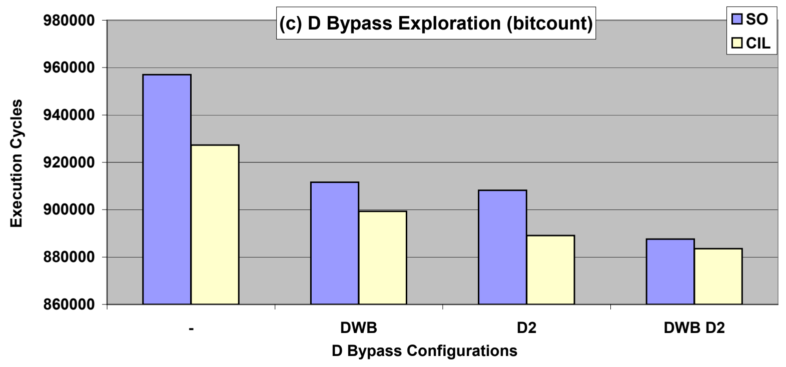


Figure 2.24: D-bypass Exploration for bitcount benchmark

Figure 2.24 and Figure 2.25 focuses on varying D and M bypasses while keeping the rest in-place. Figure 2.24 shows that the simulation-only exploration evaluates the performance of the bypass configurations with one bypass as equivalent. However, our PBExplore determines that if you can have only one bypass, the bypass from the D2 pipeline stage is a superior choice. We make similar observations for the M-bypass exploration in Figure 2.25.

Thus, performance evaluation by the simulation-only exploration and our bypass-sensitive compiler-in-the-loop exploration differ significantly, and may lead to different design decisions.

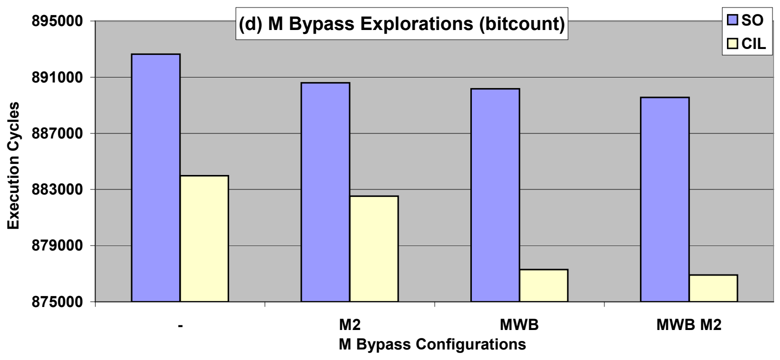


Figure 2.25: M-bypass Exploration for bitcount benchmark

2.11.4 Compiler-in-the-Loop Exploration

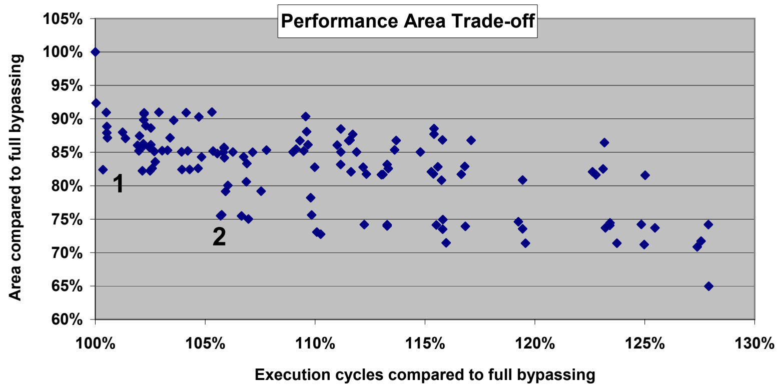


Figure 2.26: Power-Area trade-offs using PBExplore

To demonstrate that PBExplore can effectively perform a multi-dimensional exploration, we vary the bypasses only for the first operand, and assume that all the bypasses reach the other two operands. Thus there are $2^7 = 128$ bypass configurations. Figure 2.26 shows the performance-area evaluation of each bypass configurations computed using PBExplore. Similarly, Figure 2.27 shows the performance-energy evaluation of each bypass configurations computed using PBExplore.

The performance area and energy consumption are shown relative to that of a fully bypassed processor. The interesting pareto-optimal design points 1 and 2 are marked in both the graphs. Design point 1 represents the bypass configuration when MWB and XWB do not bypass to the first operand. This bypass configuration, uses 18% less area than full bypassing and consumes 14% less energy than full bypassing, while suffering only 2% performance penalty. Similarly design point 2 represents the bypass configuration when only

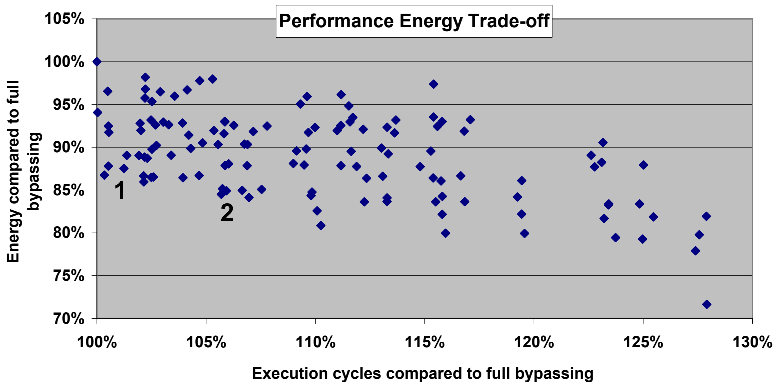


Figure 2.27: Power-Energy trade-offs using PBExplore

D2 and X2 bypass to the first operand. This configuration uses 25% less area and consumes 16% less power than fully bypassed processor, while losing only 6% on performance. These configurations represent cheaper (in area and energy consumption) design alternatives, at the cost of minimal performance degradation. These are exactly the kind of trade-offs that an embedded processor designers would need to evaluate when customizing bypasses.

2.12 Summary

The design of a processor's pipeline has a profound impact on the power, performance of the processor; consequently it is a very important step in processor design. Significant research efforts have been invested in developing compiler techniques to exploit processor pipeline details in order to improve the performance, energy consumption etc.

In this chapter we advance the existing architecture-sensitive compiler technology by making the compiler sensitive to the bypasses present in the processor pipeline. Bypasses are present in most pipelined processors, as they eliminate certain data hazards and improve performance. However, bypasses often have significant impact on the cycle-time, power consumption and the complexity of the processor. Embedded system designers, in their quest to achieve all the multi-dimensional design goals in chorus, want to customize the bypasses. They want to keep only the most useful bypasses, and get rid of the less useful ones, with minimum loss in performance. Such a processor, in which only some of the bypasses are present, is called a partially bypassed processor. Partially bypassed processors pose a challenge for code generation. Missing bypasses cause data hazards and result in performance loss. Existing architecture-sensitive compilation techniques cannot model partial bypasses and are therefore not able to accurately detect all pipeline hazards in a partially bypassed processor.

In this chapter we proposed the Operation Tables representation to model a partially bypassed processor pipeline. Our Operation Table based compiler is able to accurately detect all kinds of pipeline hazards in a partially bypassed processor, and is therefore able to generate code for any given bypass configuration. On the base configuration of the partially bypassed Intel XScale processor, our bypass-sensitive compiler is able to generate up to 20% better performing code than a bypass insensitive compiler.

Further, we developed a CIL DSE to explore bypasses in a processor pipeline. Traditionally the decision of which bypasses to keep and which ones to remove was done using the designer's intuition and/or a SO DSE. We show that the performance evaluations of bypass configurations as done by SO DSE are not only significantly different than the results of the CIL DSE, but there is a difference in trends too. This implies that SO DSE can result in sub-optimal design decisions and therefore inferior processor microarchitecture, establishing the need and usefulness of our CIL DSE strategy in designing the bypasses of the processor pipeline.

While this approach is useful for both general purpose and embedded processors, we believe it is particularly important for embedded processor designs that need to meet strict timing, area, and codesize constraints. Further, in this chapter we have demonstrated the need and usefulness of CIL DSE for the design of partial bypassing in pipelined processors. However, we believe CIL DSE is useful in designing many other architectural features at the processor pipeline design abstraction.

Chapter 3

Instruction Set Architecture Design

The instruction set of a processor determines the syntax and the semantics of the instructions of the processor. As shown in Figure 3.1, traditionally the compiler is not aware of the internals of the processor, and the instruction set is the only interface it understands. In its most basic form, a compiler just recodes an application in a high-level programming language (e.g., C, C++, Java) into the binary instructions of the processor. Thus the instruction set is the only interface between the application code and the processor; it provides the mechanisms for the application to exercise the internal components of the processor.

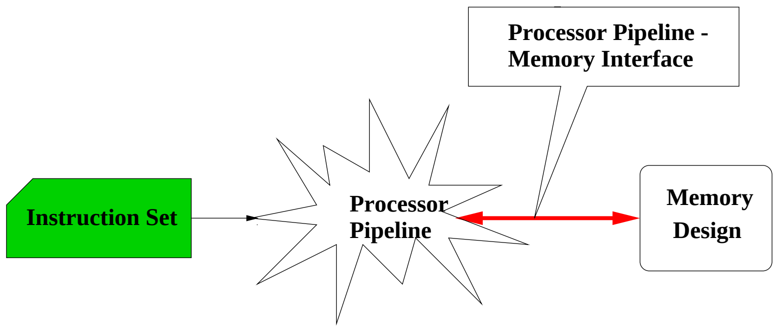


Figure 3.1: Instruction Set Architecture Design Abstraction

Among generations of processors, the instruction set is often updated to better exploit the new processor design internals. For example, the ARM instruction set has gone through 11 revisions and may go through several more. The instruction set design is very crucial and has tremendous impact on the power, performance etc. achievable by the processor. Every change in the instruction set requires reevaluation of the applications to determine if it can make efficient use of the processor design via the interface.

Typically the decisions of the changes in the instruction set are manual. To evaluate the effectiveness of the changes, or evaluate the new instruction set, designers typically hand generate the code and measure the power, performance metrics etc. of the processor. This method of evaluating the new instruction set may be okay if finally humans will be developing code for the processor. However due to explosion in the size of the software executing on embedded processors, this is no longer feasible. Increasingly compilers are now becoming the default code generators. In such a case, evaluating the goodness of a new instruction set by hand compiling the application may not be accurate.

To remove this limitation of hand generating code to evaluate a new architectural feature, Architecture Description Langugage (ADL) based exploration techniques have been proposed. In these semi-automated methods, the instruction set of the processor is described in the ADL, and then a compiler/simulator toolchain is generated for the described instruction set architecture. Different instruction set architectures can be evaluated by modifying the ADL, and estimating the power, performance etc. of the new instruction set architecture using the generated compiler and simulator. The key to these approaches lie in the instruction-set level retargetability of the generated compiler. For instance, the Instruction Set Description Language (ISDL) describes the instruction-set of processor architectures. ISDL is used by the Aviv compiler [24] to provide instruction set retargetability. Similarly in the nML ADL a hierarchical scheme is used to describe instruction sets. nML has been used by code generators CBC and CHESS [25] to provide instruction-set retargetability. The EXPRESS compiler of the EXPRESSION ADL [3] also provides instruction-set level retargetability.

In this chapter we focus on developing a novel compilation technique to achieve high degrees of code compression using a popular instruction set architecture feature, called rISA. Next, we demonstrate the need and usefulness of CIL DSE exploration using this compilation technique while designing the rISA architectural feature.

3.1 reduced bitwidth Instruction Set Architecture (rISA)

Programmable RISC processors are increasingly being used to design modern embedded systems. Examples of such systems include cell-phones, printers, modems, handhelds etc. Using RISC processors in such systems offers the advantage of increased design flexibility, high computing power and low on-chip power consumption. However, RISC processor systems suffer from the problem of poor code density which may require more ROM for storing program code. As a large part of the IC area is devoted to the ROM, this is a severe limitation for large volume, cost sensitive embedded systems. Consequently, there is a lot of interest in reducing program code size in systems using RISC processors.

Traditionally, ISAs have been fixed width (e.g., 32-bit SPARC,64-bit Alpha) or variable width (e.g., x86). Fixed width ISAs give good performance at the cost of code size and variable width ISAs give good performance at the cost of added decode complexity. Neither of the above are good choices for embedded processors where performance, code size, power, all are critical constraints. Dual width ISAs are a good tradeoff between code size flexibility and performance, making them a good choice for embedded processors. Processors with dual with ISAs are capable of executing two different Instruction-Sets. One is the "normal" set, which is the original instruction set, and the other is the "reduced bit-width" instruction set that encodes the most commonly used instructions using fewer bits. A very good example is the ARM [26] ISA with a 32-bit "normal" Instruction Set and a 16-bit Instruction Set called "Thumb". Other processors with a similar feature include the MIPS 32/16 bit TinyRISC [27], ST100 [28] and the Tangent A5 [29]. We term this feature as the "**r**educed bit-width **I**nstruction **S**et **A**rchitecture" (**rISA**).

Processors with the rISA feature dynamically translate (or decompress, or expand) the narrow rISA instructions into corresponding normal instructions. This translation usually occurs before or during the decode stage. Typically, each rISA instruction has an equivalent instruction in the normal instruction set. This makes translation simple and can usually be done with minimal performance penalty. As the translation engine converts rISA instructions into normal instructions, no other hardware is needed to execute rISA instructions. If the whole program can be expressed in terms of rISA instructions, then up to 50% code size reduction can be achieved.

The fetch-width of the processor being the same, the processor when operating in rISA mode fetches twice as many rISA instructions (as compared to normal instructions) in each fetch operation. Thus while executing rISA instructions, the processor needs to make lesser fetch requests to the instruction memory. This results in a decrease in power and energy consumption by the instruction memory subsystem.

Typically, each rISA instruction has an equivalent instruction in the normal instruction set. This makes the translation from rISA instructions to normal instructions very simple. Research e.g., [30], [31] has shown that the translation unit for a rISA design can be very small and can be implemented in a fast, power efficient manner. Furthermore, we have shown that significant energy savings achieved by executing rISA code [32].

Thus, the main advantage of rISA lies in achieving low code size and low energy consumption with minimal hardware alterations. However since more rISA instructions are required to implement the same task, rISA code has slightly lower performance compared to the normal code.

3.2 rISA Architecture

A **rISA** processor is one which supports instructions from two different Instruction Sets. One is the "normal" 32-bit wide Instruction Set, and the other is the "narrow" 16-bit wide Instruction Set. The "narrow" instructions comprise the **r**educed bit-width **I**nstruction **S**et **rIS**. The code for a rISA processor contains both normal and rISA instructions, but the processor dynamically converts the rISA instructions to normal instructions, before or during the instruction decode stage.

3.2.1 rISA: Software Aspects

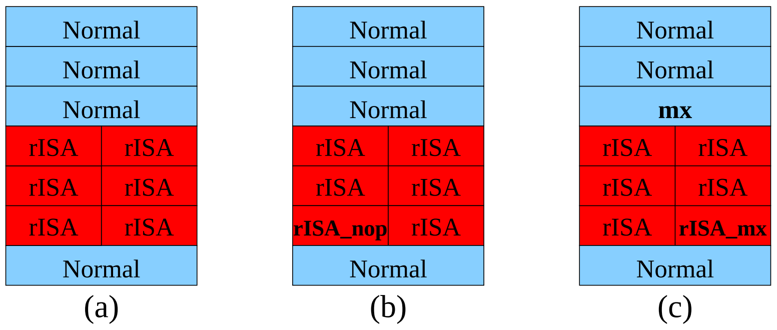


Figure 3.2: Normal and rISA instructions co-exist in Memory

3.2.1.1 Adherence to Word Boundary

The code for rISA processors contains instructions from both the instruction sets, as shown in Figure 3.2 (a). Many architectures impose the restriction that all code should adhere to the word boundary.

In order for the normal instructions to adhere to the word boundary, there can be only even number of contiguous rISA instructions. To achieve this, a rISA instruction that does not change the state of the processor is needed. We term such an operation as *rISA_nop*. The compiler can then pad odd-sized sequence of rISA instructions with *rISA_nop*, as shown in Figure 5 (b). ARM-Thumb and MIPS32/16 impose such architectural restrictions, while the ARC processor can decode the instructions even if they cross the word boundary.

3.2.1.2 Mode Change Instructions

rISA processors operate in two modes, the normal mode and the rISA mode. In order to dynamically change the execution mode of a processor, there should be a mechanism in software to specify change in execution mode. For most rISA processors, this is accomplished using explicit mode change instructions. We term an instruction in the normal instruction set that changes mode from normal to rISA the *mx* instruction, and an instruction in the rISA instruction set that changes mode from rISA to normal the *rISA_mx* instruction. The code including the mode change instructions is shown in Figure 3.2 (c).

In ARM/Thumb, ARM instructions *BX* or *BLX* switch the processor to Thumb mode. The ARM *BX* instruction is a version of a ARM branch instruction, which changes the execution mode of the processor to rISA mode. Similarly the ARM *BLX* instruction is a version of ARM *BL* instruction (Branch and Link), with the additional functionality of switching the processor to rISA mode. Similar earmarked instructions also exist in the Thumb instruction set to switch the processor back to the normal mode of operation. The MIPS16 ISA has an interesting mechanism for specifying mode changes. All routines encoded using MIPS16 instructions begin at the half word boundary. Thus, calls (and returns) to half word aligned addresses change the mode from normal to rISA. The ARC Tangent A5 processor on the other hand allows native execution of the ARCompact instructions. No special instruction is required to switch modes.

3.2.2 rISA: Hardware Aspects

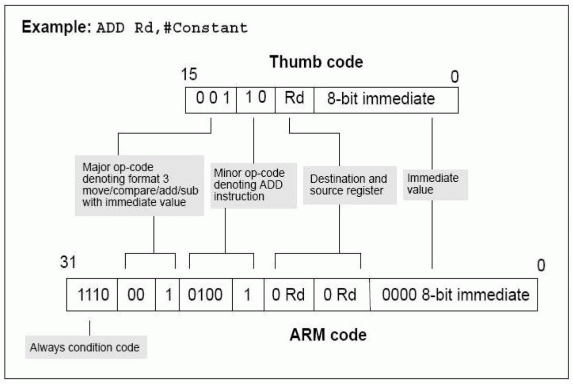

Example: ADD Rd,#Constant
Thumb code
15
0
0 0 1
1 0
Rd
8-bit immediate
Major op-code denoting format 3 move/compare/add/sub with immediate value
Minor op-code denoting ADD instruction
Destination and source register
Immediate value
31
0
1110
00
1
0100
1
0 Rd
0 Rd
0000 8-bit immediate
Always condition code
ARM code

Figure 3.3: Simple translation of Thumb instruction to ARM instruction

Although the code for a rISA processor contains instructions from both the Instruction Sets, the instruction fetch mechanism of the processor is oblivious of the presence of rISA instructions. The fetched code is interpreted (decoded) as normal or rISA instruction depending on the operational mode of the processor. When the processor is in rISA mode, the fetched code is assumed to contain two rISA instructions. The first one is translated into normal instruction, while the second one is latched and kept for the next cycle of execution.

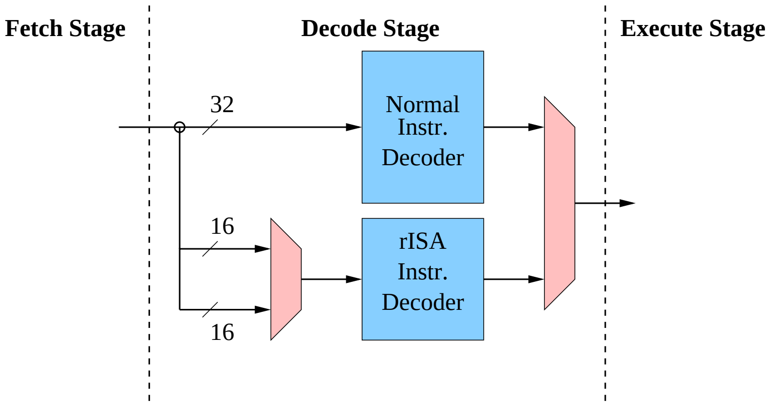


Figure 3.4: Single step Decoding of rISA instructions

Figure 3.3 shows an example of translation of a Thumb-ADD instruction to a normal ARM-ADD instruction. The translation can be realized in terms of simple and small table lookups. Since the conversion to normal instructions is done during or before the instruction decode stage, the rest of the processor remains same. To provide support for rISA typically only decode logic of the processor needs to be modified. The rISA instructions can be decoded by either single step decoding or two step decoding.

Figure 3.4 shows the one step decoding of rISA instructions. The single step decoding of rISA instructions is just like performed at the same time as the decoding of the normal instructions. Single step decoding is typically simpler because of the rISA instructions are “narrow”.

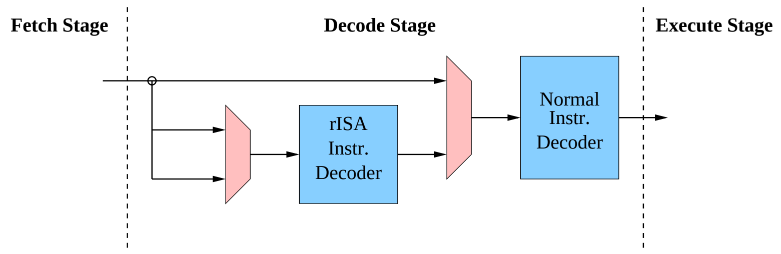


Figure 3.5: Two step Decoding of Thumb instructions

In the two step decoding shows in Figure 3.5, the rISA instructions are first translated to normal instructions. The normal instructions can then be decoded as before. Although such implementation requires minimal logical changes to the existing architecture, it may lengthen the cycle time. ARM7TDMI implements two step decoding approach of Thumb instructions.

3.3 Related Work

Many leading 32-bit embedded processors also support the 16 bit (rISA) instruction set to address both memory and energy consumption concerns of the embedded domain [26], [27], [28], [29].

There has been previous research effort to achieve further code compression with the help of architectural modifications to rISA. The ARM Thumb IS was redesigned by Kwon et al. [33] to compress more instructions and further improve the efficiency of code size reduction. This new Instruction Set is called Partitioned Register Extension (PARE), reduces the width of the destination field and uses the saved bit(s) for the immediate addressing field. The register file is split into (possibly overlapping) partitions, and each 16-bit instructions can only write to a particular partition. This reduces the number of bits required to specify the destination register. With a PARE-aware compiler, the authors claim to have achieved a compression ratio comparable to Thumb and MIPS16.

Another direction of research in rISA architectures has been to overcome the problem of decrease in performance of rISA code. Kwon et al. [34] proposed a parallel architecture for executing rISA instructions. TOE(Two Operation Execution) exploits Instruction Level Parallelism provided by the compiler. In the TOE architecture all rISA instruction occur in pairs. With 1-bit specifying the eligibility of the pair of rISA instructions to execute in parallel, the performance of rISA can be improved. Since the parallelization is done by the compiler, the hardware complexity remains low.

Krishnaswamy et al. [35] observed that there exist Thumb instruction pairs that are equivalent to single ARM instructions throughout the 16-bit Thumb code. They enhanced the Thumb instruction set by an AX (Augmenting Extensions) instructions. Compiler finds the pairs of Thumb instructions that can be safely executed as single ARM instruction, and replace them by AX+Thumb instruction pairs. They coalesce the AX with the immediately following Thumb instruction at decode time and generate an ARM instruction to execute single instruction, thus increasing performance.

While there has been considerable research in the design of architectures/architectural features for rISA, the compiler techniques employed to generate code targeted for such architectures are rudimentary. Most existing compilers either rely on user guidance or perform a simple analysis to determine which routines of the application to code using rISA instructions. These approaches, which operate at the routine level granularity, are unable to recognize opportunities for code size optimization within routines. We propose instruction-level granularity of rISAization and present compiler framework to generate optimized code for such rISA architectures. Our technique, is able to aggressively reduce code size by discovering codesize reduction opportunities inside a routine, resulting in high degrees of code compression.

3.4 Challenges in Compilation for rISA Architectures

Although up to 50% code compression can be achieved using rISA, owing to severe restrictions on the number of operations, register accessibility and reduced functionality, it is in practice tough to consistently achieve more than 30% code size reduction. In order to alleviate such severe constraints, several solutions have been proposed. We discuss several such architectural features in the light of aiding code generation in rISA processors.

3.4.1 Granularity of rISAization

We term the compiler-process of converting normal instructions into rISA instructions as **rISAization**. Existing compilers like the ARM-Thumb compiler (as yet) supports the conversion at a routine level granularity. Thus, all the instructions in a routine can be in exactly one mode, the normal ARM mode, or the Thumb mode. A routine cannot have instructions from both the ISAs. Furthermore existing Compilers rely on human analysis to determine which routines to implement in ARM instructions, and which ones in Thumb instructions.

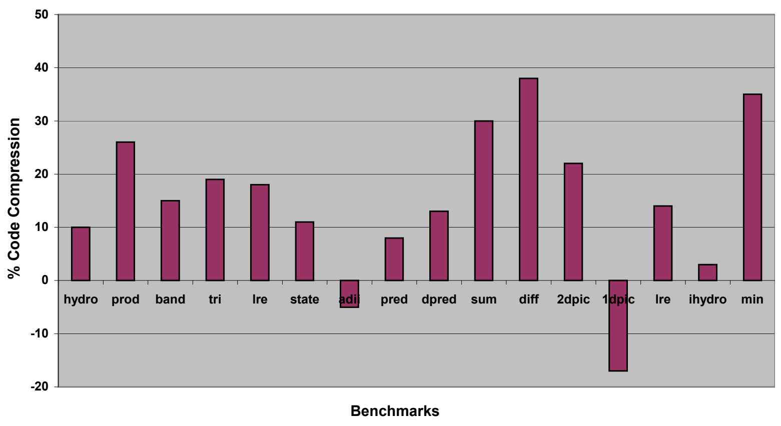


Figure 3.6: Code compression achieved by routine-level rISAization

Figure 3.6 plots the code compression achieved by MIPS32/16 compiler performing indiscriminate rISAization. The compiler could achieve 38% code compression on *diff* benchmark, that has only one routine and low register pressure. All the instructions could be mapped to rISA instructions. However on *1dpic*, the most important routine had high register pressure, that resulted in register spilling and thus leading to an increase in code size. Similar is the case with *adii*. Some other benchmarks had routines in which the conversion resulted in an increase in the code size. Thus although routine-level granularity of rISAization can achieve high degrees of code compression on small routines, it is unable to achieve decent code compression on application level. In fact MIPS32/16 compiler could achieve only 15% code size reduction on our set of benchmarks. This is because not a lot of routines can be found, whose conversion results in substantial code compression. The inability to compress code is due to two main reasons (i) rISAization may result in an increase in code size. (ii) rISAization of routines that have high register pressure results in register spills and thus increase in the code.

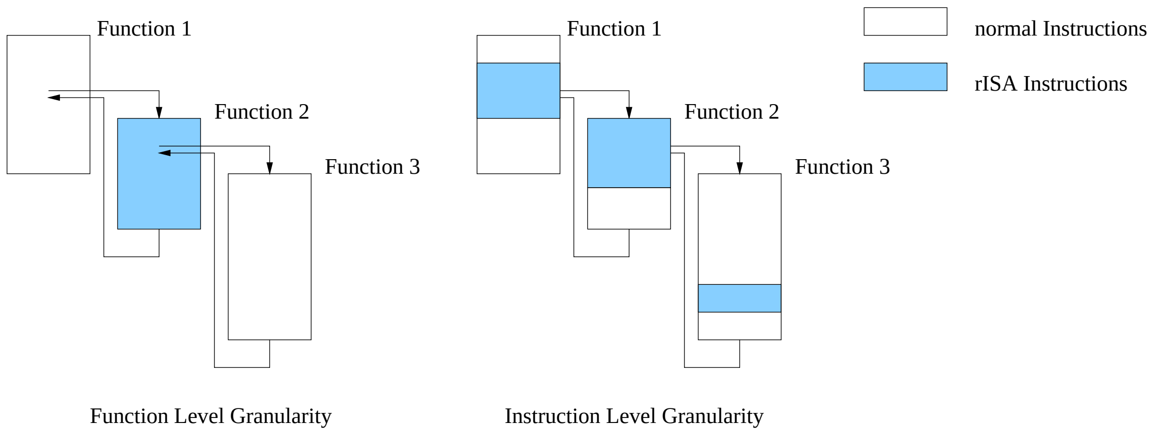


Figure 3.7: Routine-level Granularity versus Instruction-level Granularity

Figure 3.7 explains the two major drawbacks of rISAizing at routine-level granularity and shows that instruction-level granularity alleviates these drawbacks.

First is that a routine-level granularity approach misses out on the opportunity to rISAize code sections inside a routine, which is deemed non profitable to rISAize. It is possible that it is not profitable to rISAize a routine as a whole, but some parts of it can be profitably rISAized. For example, the *Function 1 and Function 3* are found to be non-profitable to rISAize as a whole. Traditional routine-level granularity approaches will therefore not rISAize these routines, while instruction-level granularity approaches will be able to achieve some code size reduction by identifying and rISAizing only some profitable portions of the routine.

Second, the compiler is not able to leave out some regions of code inside a routine that may incur several register spills. It is possible that leaving out some pieces of code inside a profitable routine may increase the code compression achieved. For example, in Figure 3.7, the instruction-level granularity approaches have the choice to leave out some regions of code inside a routine to achieve higher code compression.

Thus consistently high degree of code compression can be achieved by rISAization at instruction level granularity. However, instruction-level granularity of rISAization has some overheads. Instruction level granularity of rISAization needs explicit mode change instructions. We define a normal instruction *mx* changes the execution mode of the processor from normal to rISA, while a rISA instruction *rISA_mx* changes the execution mode of the processor from rISA mode to normal mode. The mode change instructions have to be inserted in the code at the boundaries of normal and rISA instructions. Unlike the conversion at routine level granularity, this causes an increase in code size. But our results show that even with this code size penalty, consistently higher degrees of code compression can be achieved by rISAization at instruction level granularity.

3.4.2 rISA Instruction Set

The rISA instruction set is tightly constrained by the instruction width. Typical instruction sets have three operand instructions; two source operands and one destination operand. Only 16 bits are available to encode the opcode field and the three operand fields. The rISA design space is huge, and several instruction set idiosyncrasies makes it very tough to characterize. In informal terms if *rISA_wxyz* defines a rISA instruction set with opcode width w-bits, and three operands widths as x-bit, y-bit, and z-bits respectively. Most interesting rISA instruction sets are bound by *rISA_7333* and *rISA_4444*.

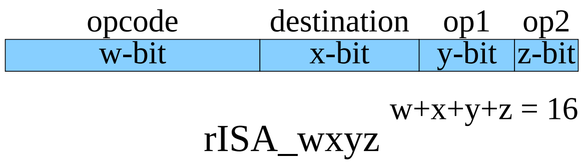


Figure 3.8: rISA Instruction Format

The *rISA_7333* format describes an instruction set in which the opcode field is 7-bit wide, and each operand is 3-bit wide. Such an instruction set would contain 128 instructions, but each instruction can access only 8 registers. Although such a rISA instruction set can rISAize large portions of code, but register pressure may become too high to achieve a profitable encoding. On the other extreme is the *rISA_4444* format, which has space for only 16 instructions, but each instruction can access up to 16 registers. For applications that do not use a wide variety of instructions, but have high register pressure, such a rISA instruction set is certainly a good choice.

The design space between the two extremes is huge. All realistic rISA instruction sets contain a mix of both type of instructions, and try to achieve the "best of both worlds". Designing a rISA instruction set is a essentially a trade-off between encoding more instructions in the rISA instruction set, and providing rISA instructions access to more registers.

The "implicit operand format" of rISA instructions is a very good example of the trade-off the designers have to make while designing rISA. In this feature, one (or more) of the operands in the rISA instruction is hard-coded (i.e. implied). The implied operand could be a register operand, or a constant. In case a frequently occurring format of add instruction is *add* R_i R_i R_j (where the first two operands are the same), a rISA instruction *rISA_add1* R_i R_j, can be used. In case an application that access arrays produces a lot of instructions like $addr = addr + 4$ then a rISA instruction *rISA_add4 addr* which has only one operand might be very useful. The translation unit, while expanding the instruction, can also fill in the missing operand fields. This is a very useful feature that can be used by the compiler to generate good quality code. ARC Tangent A5 processor uses this feature extensively to optimize ARCompact instruction set.

3.4.3 Limited Access to Registers

rISA instructions usually have access to only a limited set of processor registers. This results in increased register pressure in rISA code sections. A very useful technique to increase the number of useful registers in rISA mode is to implement a *rISA_move* instruction that can access all registers. This is possible because a move operation has only two operands and hence has more bits to address each operand.

3.4.4 Limited width of immediate operands

A severe limitation of rISA instructions is the inability to incorporate large immediate values. For example, with only 3 bits are available for operands, the maximum unsigned value that can be expressed is 7. Thus, it might be useful to vary the size of the immediate field depending on the application and the values that are (commonly) generated by the compiler. Increasing the size of the immediate fields however, reduces the number of bits available for opcodes (and also the other operands). Several architectures implement a *rISA_extend* instruction, which extends the immediate field in the next rISA instruction. Such an instruction is very useful to be able to rISAize contiguous large portions of code.

3.5 Our Compilation for rISA Architectures

We implemented our rISA compiler technique in the EXPRESS retargetable compiler. EXPRESS [2] is an optimizing, memory-aware, Instruction Level Parallelizing (ILP) compiler. EXPRESS uses the EXPRESSION ADL [3] to retarget itself to a wide class of processor architectures and memory systems. The inputs to EXPRESS are the application specified in C, and the processor architecture specified in EXPRESSION. The front-end is GCC based and performs some of conventional optimizations. The core transformations in EXPRESS include **RDLP** [36] – a loop pipelining technique, **TiPS** : Trailblazing Percolation Scheduling [37] – a speculative code motion technique, Instruction Selection and Register Allocation. The back-end generates assembly code for the processor ISA.

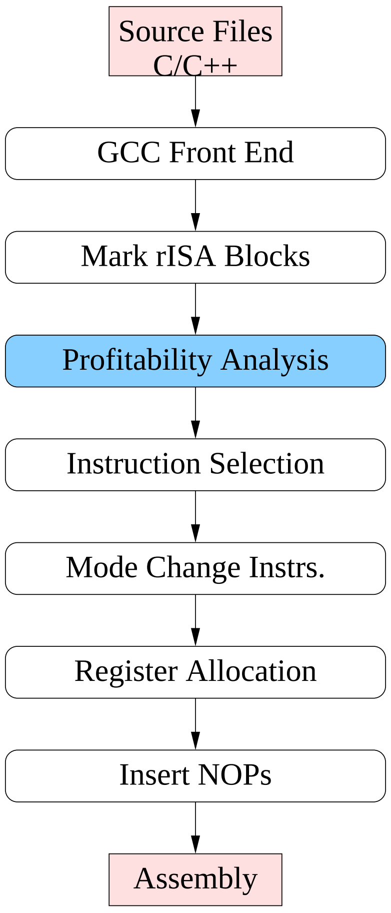


Figure 3.9: rISA Compiler Steps

A rISA compiler not only needs the ability to selectively convert portions of application into rISA instruction, but also heuristics to perform this conversion only where it is profitable. Figure 3.9 shows the phases of the EXPRESS compiler with our rISAization technique. The code generation for rISA processors in EXPRESS is therefore a multi-step process:

3.5.1 Mark rISA Blocks

Various restrictions on the rISA instruction set, means that several normal instructions may not be convertible into rISA instructions. For example, an instruction with a large immediate value may not be rISAizable. The first step in compilation for a rISA processor is thus marking all the instructions that can be converted into rISA instructions. However, converting all the marked instructions into rISA instructions may not be profitable, because of the overhead associated with rISAization. The next step is therefore, to decide which contiguous list of marked instructions are profitable to convert to rISA instructions. Note that a list of contiguous marked instructions can span across basic block boundaries. To ensure correct execution, mode change instructions need to be added, so that the execution mode of processor (normal or rISA) matches that of the instructions it is executing. The instruction selection, however is done within basic blocks. Contiguous list of marked instructions in a basic block is termed **rISABlock**.

3.5.2 Profitability Heuristic for rISAization

Even though all the instructions in a rISABlock, can be rISAized, it may not be profitable (in terms of code compression, or performance) to rISAize the rISABlock. For example, if a rISABlock is very small, then the mode change instruction overhead could outshine any code compression achievable by rISAization. Similarly if the rISABlock is very big, the increase register pressure (and register spilling therefore) could make rISAization the rISABlock a bad idea. Thus an accurate estimation of code size and performance trade-off is necessary before rISAizing a rISABlock. In our technique, the impact of rISAization on code size and performance is estimated using a profitability analysis (PA) function. The PA function estimates the difference in code size (CS) and performance (PF) if the block were to be implemented in rISA mode as compared to normal mode. The compiler (or user) can then use these estimates to trade-off between performance and code size benefits for the program. Next we describe how the PA function measures the estimated impact on code size and performance.

```
Estimate_CodeSize_Reduction
01: CS1 = $sizeof(mx) + sizeof(rISA\_mx)$
02: CS2 = $sizeof(rISA\_nop)$
03: CS3 = $Extra\_Spill\_Reload\_Estimate(bl)$
04: return $sizeBlock(bl, NORMAL) - sizeBlock(bl, rISA) - CS1 - CS2 - CS3$

Extra_Spill_Reload_Estimate(Block bl)
05:   // Estimate spill code if the block is rISAized
06:   $extra\_rISA\_reg\_press = Avg\_Reg\_Press(bl, rISA\_vars) - K1 \times rISA\_REGS$
07:   if $(extra\_rISA\_reg\_press > 0)$
08:     $avail\_non\_rISA\_regs = TOTAL\_REGS - rISA\_REGS$
09:     $rISA\_spills = \frac{extra\_rISA\_reg\_press \times bl.num\_instrs}{Avg\_Live\_Len(bl)}$
10:   else
11:     $avail\_non\_rISA\_regs = TOTAL\_REGS - rISA\_REGS - extra\_rISA\_reg\_press$
12:     $rISA\_spills = 0$
13:   endIf

14:   $extra\_non\_rISA\_reg\_press = Avg\_Reg\_Press(bl, non\_rISA\_vars)$
15:                               $-K1 \times avail\_non\_rISA\_regs$
16:   if $(extra\_non\_rISA\_reg\_press > 0)$
17:     $non\_rISA\_spills = extra\_non\_rISA\_reg\_press$
18:   else
19:     $non\_rISA\_spills = 0$
20:   endIf

21:   spill_code_if_rISA = $rISA\_spills \times SIZE\_rISA\_INSTR$
22:                        $+non\_rISA\_spills \times SIZE\_NORMAL\_INSTR$

23:   // Estimate spill code if the block is NOT rISAized
24:   $extra\_normal\_reg\_press = Avg\_Reg\_Press(bl, all\_vars) - K1 \times TOTAL\_REGS$
25:   if $(extra\_normal\_reg\_press > 0)$
26:     $normal\_spills = \frac{extra\_normal\_reg\_press \times bl.num\_instrs}{Avg\_Live\_len(bl)}$
27:   else
28:     $normal\_spills = 0$
29:   endIf

30:   spill_code_if_normal = $normal\_spills \times SIZE\_NORMAL\_INSTR$
31:   extra_spill_code = $spill\_code\_if\_rISA - spill\_code\_if\_normal$
32:   extra_reload_code = $K2 \times extra\_spill\_code \times Avg\_Uses\_Per\_Def$
33:   return $extra\_spill\_code + extra\_reload\_code$

Parameters
TOTAL_REGS = 16, rISA_REGS = 8
SIZE_rISA_INSTR = 2 bytes, SIZE_NORMAL_INSTR = 4 bytes
K1 and K2 are control constants
```

Figure 3.10: Profitability heuristic for rISAization

3.5.2.1 Code Size (CS)

Figure 3.10 shows the portion of the PA function that estimates the code size reduction due to rISA. Ideally, converting a block of code to rISA instructions reduces the size of the block by half. However, the conversion typically incurs an overhead that reduces the amount of compression. This overhead is composed of three factors:

Mode Change Instructions (CS1): Before every block of rISA instructions, a *mx* (Mode Change from normal to rISA) instruction is needed. This causes an increase in code size by one full length instruction. At the end of every rISA block, a *rISA_mx* (Mode Change from rISA to normal) instruction is needed, causing an increase in code size by the size of the rISA instruction. Thus for an architecture with normal instruction length of 4 bytes and rISA instruction of 2 bytes, $CS1 = 4 + 2 = 6$bytes.

NOP (CS2): Most architectures require that normal instructions be aligned at word boundaries. However, rISAizing a block with odd number of instructions[1] will cause the succeeding normal instruction to be mis-aligned. In such cases, an extra *rISA_nop* (No-operation instruction) needs to be added inside the rISA block. We conservatively estimate that each rISA block needs a rISA_nop instruction. $CS2 = 2$bytes.

Spills/Reloads (CS3): Due to limited availability of registers, rISAizing a block may require a large amount of spilling (either to memory or to non-rISA registers). As this greatly impacts both code size and performance it is important to accurately estimate the number of spills (and reloads) due to rISAization. The PA function estimates the number of spills and reloads due to the rISA block by calculating the average register pressure[2] due to the variables in the block.

The first step is to calculate the amount of spill code inserted if the block is rISAized (line 20 in Figure 3.10). The block may contain variables that need to be allocated to the rISA register set and variables that can be allocated to any registers. Thus, rISA spill code is estimated as the total of spills due to rISA variables (lines 05-13) and spills due to non rISA variables (lines 14-19). The constant *K1* can be used to control the importance of spill code in estimation.

The function *Avg_Reg_Press* returns the average register pressure for variables of a particular type (rISA or non rISA) in a block. The function *Avg_Live_Len* returns the average distance between the definition of a variable in a block and its last use (i.e. its life-time). In a block, the extra register pressure (that causes spilling) is the difference between Avg_Reg_Press and the number of available registers (lines 06, 14, 22). Each spill reduces the register pressure by 1 for the life time of the variable. So, a block with size

[1]Including the rISA_mx instruction

[2]Register Pressure is defined as the number of variables live at the point in the program.

num_instrs requires *num_instrs/Avg_Live_Len* spills to reduce the register pressure by 1. Thus, the number of spills required to mitigate the register pressure is equal to the extra register pressure multiplied by the number of spills required to reduce register pressure by 1 (lines 09, 16, 24).

The next step is to estimate the total number of spills if the block is not converted to rISA instructions (line 28). This is accomplished in a manner similar to that of estimation of rISA variables.

As each spill also requires reloads to bring the variable to a register before its use, it is necessary to also calculate the number of extra reloads due to conversion to rISA. The PA function estimates the number of reloads as a factor of number of spills in the rISA Block. The constant *K2* can be used to control the importance of reload code in estimation.

The total reduction in code size of the block due to rISAization (line 04) is
$CS = 2 \times NumInstrs(rISABlock) - CS1 - CS2 - CS3.$
A *CS* value greater than zero implies that converting the block to rISA instructions is profitable in terms of code size.

3.5.2.2 Performance (PF)

The impact of converting a block of instructions into rISA on performance is difficult to estimate. This is especially true if the architecture incorporates a complex instruction memory hierarchy (with caches). Our technique makes a crude estimate of the performance impact based on the latency of the extra instructions (due to the spills/reloads, and due to the mode change instructions). A more accurate estimate can be made by also considering the instruction caches and the placement of the blocks in program memory.

3.5.3 Instruction Selection

EXPRESS uses a tree pattern matching based algorithm for Instruction Selection. A tree of generic instructions is converted to a tree of target instructions. In case a tree of generic instructions can be replaced by more than one target instruction tree, the one with lower cost is selected. The cost of a tree depends upon the user's relative importance of performance and code-size. Our approach towards compiling for rISA, looks at the rISA conversion as a natural part of the Instruction Selection process. The Instruction Selection phase uses a profitability heuristic to guide the decisions of which section of a routine to convert to rISA instructions, and which ones to normal target instructions. All generic instructions within profitable rISABlocks are replaced with rISA instructions and all other instructions are replaced with normal target instructions. Replacing a generic instruction with a rISA instruction involves both selecting the appropriate rISA opcode, and also restricting the operand variables to the set of rISA registers.

3.5.4 Inserting Mode Change Instructions

After Instruction Selection the program comprises of sequences of normal and rISA instructions. A list of contiguous rISA instructions may span across basic block boundaries. To ensure correct execution, we need to make sure that any possible execution path, whenever there is a switch in the instructions from normal to rISA or vice versa, there is an explicit and appropriate mode change instruction. There should be a *mx* instruction when the instructions change from normal to rISA instructions, and a *rISA_mx* instruction when the instructions change from rISA instructions to normal instructions.

If the mode of instructions change inside a basic block, then there is no choice but to add the appropriate mode change instruction at the boundary. However when the mode changes at basic block boundary, the mode change instruction can be added at the beginning of the successor basic block or at the end of the predecessor basic block. The problem becomes more complex if there are more than one successors and predecessors at the junction. In such a case, we want to add the mode change instructions so as to minimize the performance degradation. So we want to add them so that they will be executed the least. We use profile information to find out the execution counts of the basic block and then solve the optimality problem.

To further motivate the problem consider two consecutive basic blocks b_i, and b_j. Suppose b_i is the successor of b_j. Further suppose that the end mode of b_i and the start mode of b_j, is the same, then there is no need to add a mode change instruction. However if there is another execution path from basic block b_k to b_j, and the last instruction of b_k is of a different mode, then explicit mode change instructions need to be inserted.

There are two choices to insert the mode change instruction, we can either insert the mode change instruction as the last instruction of b_k, or as the first instruction of b_j. In the second solution, we will have to insert a mode change instruction as the last instruction in b_i too. Thus the first solution seems to be the winner. However if execution frequency of b_k

is greater than sum of execution frequencies of b_i and b_j, then first solution results in more increase in *Dynamic Code Size*. In general there can be many control flow edges coming in b_j, and going out of b_k, making the problem more complex.

Along every possible execution path, whenever instructions change from normal to rISA, or otherwise, there should be an explicit mode change instruction. The cost of inserting a mode change instruction is equal to the execution frequency of the basic block. The problem is thus to insert mode change instructions with least cost.

The insertion of mode change instructions is performed in two steps. If mode change occurs inside a basic block, corresponding mode change instructions are inserted at the boundary of rISA Block.

After the first step, the CFG (Control Flow Graph) can be visualized as a directed graph $G = (V, E)$, where V represents the basic blocks, and E represent the Control Flow edges as shown in Figure 3.11(a). G has two distinguished vertices, the start vertex v_0 and the end vertex v_n. Three functions are thus defined on V,

- $ExecFrequency : V- > N$ gives the execution frequency for each vertex.
- $EntryMode : V \rightarrow \{Normal, rISA\}$ gives entry mode of the basic block represented by the vertex is Normal or rISA. $EntryMode(V_i)$ is *rISA* if the first instruction of the basic block is a rISA instruction. Otherwise it is *Normal*.
- $ExitMode : V \rightarrow \{Normal, rISA\}$. $ExitMode(V_i)$ is *rISA* if the last instruction of the basic block is a rISA instruction. Otherwise it is *Normal*.

We get $ExecFrequency$ for each basic block from the profile information. The functions $EntryMode$ and $ExitMode$ are computed for each basic block.

We can switch the $EntryMode$, or $ExitMode$ of a vertex by inserting a mode change instruction at the start of the basic block, or at the end of the basic block respectively. However switching the $EntryMode$ or $ExitMode$ of the vertex v_i costs $ExecFrequency(v_i)$.

The problem of mode change instruction insertion is to find $EntryMode$ and $ExitMode$ for each vertex so that,
for each edge $(v_i, v_j) \in E$,

$ExitMode(v_i) == EntryMode(v_j)$

such that the switching cost is minimized. The switching cost essentially represents the *Dynamic Code Size*.

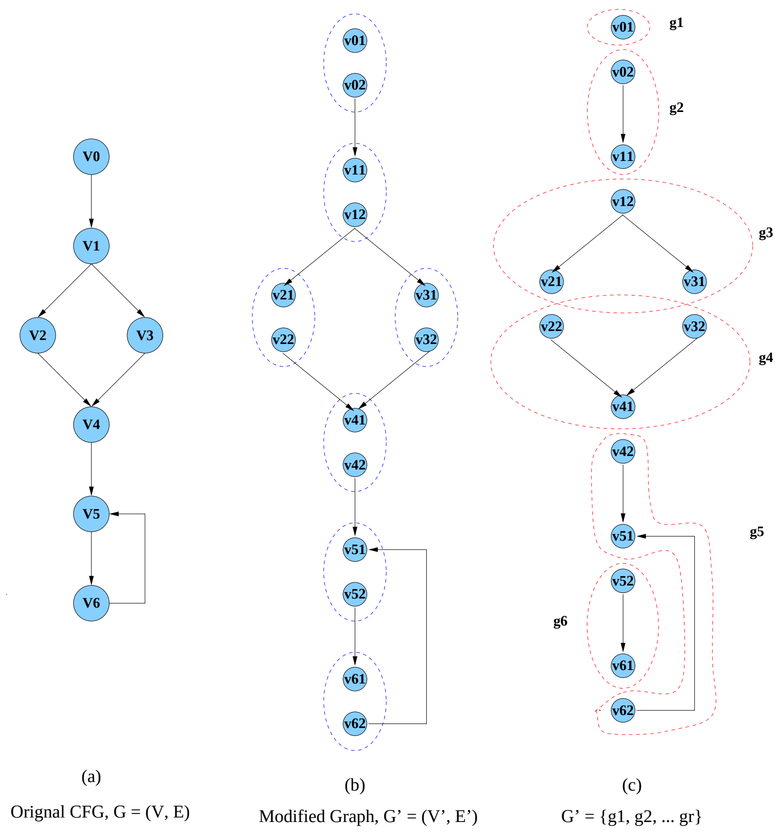


Figure 3.11: Mode change Instruction Insertion

To solve this problem we transform our graph G. We break each vertex v_i into two vertices, v_{i1} and v_{i2} in graph G' as shown in Figure 3.11(b). Vertex v_{i1} represents the entry of v_i, and v_{i2} represents the exit of v_i. All the incoming edges into v_i now come to v_{i1}, and all the outgoing edges from v_i, now emanate from v_{i2}. Two functions are defined on vertices of G',

- $ExecFrequency(v_{ij}) = ExecFrequency(v_i)$
- $Mode(v_{i1}) = EntryMode(v_i)$
- $Mode(v_{i2}) = ExitMode(v_i)$

The new Graph $G' = (V', E')$ is a forest of connected components. Our problem now reduces into finding $Mode$ for each vertex so that all the vertices in a connected component have the same mode.

We identify all the connected components of $G' = \{g_1, g_2, ...g_k\}$, as depicted in Figure 3.11(c). Each connected component is a subgraph $g_i = (V_i, E_i)$, containing a subset of

vertices,
$V_i \subset V', V_i = \{u_1, u_2, ...u_r\}$.

These vertices are partitioned into two (possibly empty) sets V_{Normal} and V_{rISA}.

$V_i = V_{Normal} \bigcup V_{rISA}$, and $V_{Normal} \bigcap V_{rISA} = \phi$.

Cost of converting all vertices to rISA Mode =
$\Sigma_{i=1..|V_{Normal}|} ExecFrequency(u_i)$.

Cost of converting all vertices to Normal Mode =
$\Sigma_{i=1..|V_{rISA}|} ExecFrequency(u_i)$.

We pick the lower cost conversion and thus decide upon the $Mode$ of each vertex in G' and hence $EntryMode$ and $ExitMode$ of each vertex in G. Finally we insert the appropriate mode change instructions. To change the $EntryMode$ of a basic block from Normal to rISA, we add a *mx* instruction as the first instruction of the basic block. To change the $EntryMode$ of a basic block from rISA to Normal, we add a *rISA_mx* instruction as the first instruction of the basic block. To change the $ExitMode$ of a basic block from Normal to rISA, we add a *mx* instruction as the last or second last instruction of the basic block. To change the $ExitMode$ of a basic block from rISA to Normal, we add a *rISA_mx* instruction as the last or second last instruction of the basic block.

Note that if the last instruction of a basic block is a branch operation, and the machine does not have a delay slot, then the mode change instruction has to be added as the second last instruction in the basic block.

3.5.5 Register Allocation

The actual register allocation of variables is done during the Register Allocation phase. The EXPRESS compiler implements a modified version of Chaitin's solution [38] to Register Allocation. Since code blocks that have been converted to rISA typically have a higher register pressure (due to limited availability of registers), higher priority is given to rISA variables during register allocation.

3.5.6 Insert NOPs

The final step in code generation is to insert *rISA_nop* instruction in *rISABlocks* that have odd number of rISA instructions. This is a straightforward step and we do not discuss about it.

3.6 Effectiveness of Our Approach

In this section, we first demonstrate the efficacy of our compilation scheme over an existing rISA architecture. We show that the register pressure based profitability function to decide which regions of code to rISAize performs good consistently. We then design several interesting rISA design points, and study the code compression obtained by using them. We show that the code compression achieved is very sensitive to the rISA design, and that a custom rISA can be designed for a set of applications to achieve high code compression.

We perform our experiments over MIPS32/16 ISA on a set of applications from numerical computation kernels (Livermore Loops), and DSP application kernels, that are often executed in embedded processors.

Our first experiment is to study the efficacy of our compilation technique. We compare the code compressions we achieve with the code compression GCC can achieve using the MIPS32/16 rISA. To this effect, we first compile the applications using GCC for MIPS32 ISA. We then compile the applications using GCC for MIPS32/16 ISA. We perform both the compilations using -Os flags with the GCC to enable all the code size optimizations. The percentage code compression achieved by GCC for MIPS16 is computed and is represented by the light bars in Figure 3.12. The MIPS32 code generated by GCC is compiled again using the register pressure based heuristic in EXPRESS. The percentage code compression achieved by EXPRESS is measured and plotted as dark bars in Figure 3.12.

It can be clearly seen from Figure 3.12 that the register pressure based heuristic performs better consistently better than GCC and successfully prevents code inflation. GCC achieves on an average 15% code size reduction, while EXPRESS achieved an average of 22% code size reduction. We used SIMPRESS[39], a cycle accurate simulator, to measure the performance impact due to rISAization. We simulated the code generated by EXPRESS on a variant of the MIPS R4000 processor that was augmented with rISA MIPS16 Instruction

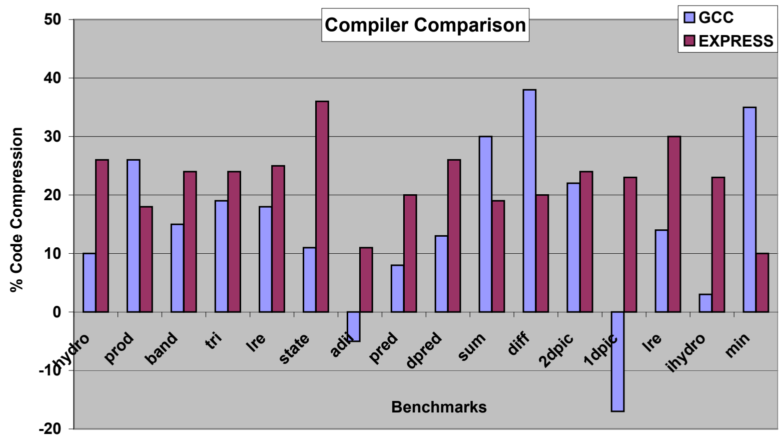


Figure 3.12: Percentage code compressions achieved by GCC and EXPRESS for MIPS32/16

Set. the memory subsystem was modeled with no caches and a single cycle main memory. the performance of MIPS16 code is on an average 6% lower than that of MIPS32 code, with the worst case being 24Thus our technique is able to reduce the code size using rISA with a minimal performance impact.

3.7 Compiler-in-the-Loop rISA Design Space Exploration

The rISA IS, because of bit-width restrictions, can encode only a subset of the normal instructions and allows access to only a small subset of registers. Such severe restrictions make the code-size reduction obtainable by using rISA very sensitive to the compiler quality and the application features. For example, if the application has high register pressure, or if the compiler does not do a good job of register allocation, it might be better to increase the number of accessible registers at the cost of encoding only a few opcodes in rISA. Thus, it is very important to perform compiler-in-the-loop design space exploration (DSE) while designing rISA architectures. Contemporary rISA processors (such as ARM/Thumb, MIPS32/16) incorporate a very simple rISA model with rISA instructions able to access 8 registers (out of 16 or 32 general-purpose registers). In this chapter, we show that varying this model by considering the application characteristics and the compiler quality results in substantial improvement of code-size reduction. We present a rISA design space exploration framework that incorporates a compiler designed to optimize for rISA and is able to explore a wide range of architecture design points.

3.7.1 rISA DSE Parameters

Conventional rISA architectures like Thumb[26] and MIPS16[40] fix the register set accessible by rISA instructions to be 8. Thus, each operand requires 3 bits for specification. This implies that, for three operand instructions, up to 128 opcodes can be encoded in rISA. The primary advantage of this approach is that most normal 32-bit instructions can also be specified as rISA instructions. However, this approach suffers from the drawback of increased register pressure possibly resulting in poor code size. One modification is to increase the number of registers accessible by rISA instructions to 16. However, in this model, only a limited number of opcodes are available. Thus, depending on the application, large sections of program code might not be implementable using rISA instructions. The design parameters that can be explored include the number of bits used to specify operands (and opcodes), and the type of opcodes that can be expressed in rISA.

Another important rISA feature that impacts the quality of the architecture is the "implicit operand format" feature. In this feature, one (or more) of the operands in the rISA instruction is hard-coded (i.e. implied). The implied operand could be a register operand, or a constant. In case a frequently occurring format of add instruction is $add\ R_i\ R_i\ R_j$ (where the first two operands are the same), a rISA instruction $rISA\_add1\ R_i\ R_j$, can be used. In case an application that access arrays produces a lot of instructions like $addr = addr + 4$ then a rISA instruction $rISA\_add4\ addr$ which has only one operand might be very useful. The translation unit, while expanding the instruction, can also fill in the missing operand fields. This is a very useful feature that can be used by the compiler to generate good quality code.

A severe limitation of rISA instructions is the inability to incorporate large immediate values. For example, with only 3 bits available for operands, the maximum unsigned value that can be expressed is 7. Thus, it might be useful to vary the size of the immediate field, depending on the application and the values that are (commonly) generated by the compiler. Increasing the size of the immediate fields will, however, reduce the number of bits available for opcodes (and also the other operands). This trade-off can be meaningfully made only with a compiler-in-the-loop DSE framework.

Various other design parameters such as partitioned register files, shifted/padded immediate fields, etc. also should be explored in order to generate a rISA architecture that is tuned to the needs of the application and to the compiler quality. While some of these design parameters have been studied in a limited context, there has been no previous work that seeks to generate rISA designs that combine all of these features. Our exploration framework, is able to quantify the impact of these features both individually and in various combinations. Also, since it incorporates the compiler, the quality of each design point is measured accurately.

3.7.2 rISA Exploration Framework

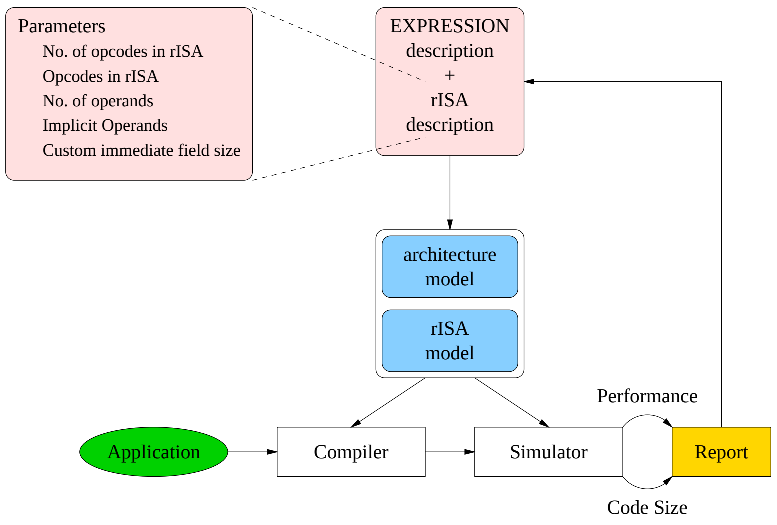


Figure 3.13: Design Space Exploration flow

Figure 3.13 presents the DSE framework used to explore rISA design points. The processor architecture (with the desired rISA features) is described using an Architecture Description Language (ADL) called EXPRESSION [3]. This description is then input to the EXPRESS retargetable compiler [2] and SIMPRESS simulator [39]. The desired applications are then compiled, simulated and the code size and performance numbers are generated for analysis. The various rISA design parameters mentioned in the previous section can be described in EXPRESSION which is then used to retarget the EXPRESS compiler to produce code optimized for those features. Below, we describe our DSE framework in greater detail. First, we describe how we capture rISA information in the EXPRESSION ADL. Then, we describe the rISA optimization techniques incorporated by the EXPRESS compiler.

3.7.3 ADL based DSE

EXPRESSION is an ADL designed to support design space exploration of a wide class of processor architectures. EXPRESSION contains an integrated specification of both structure and behavior of the processor-memory system. The structure is specified as a net-list of components (i.e., units, storages, ports and connections) along with a high-level description of the pipeline and data-transfer paths in the architecture. The behavior describes the Instruction Set of the processor. Each instruction is defined in terms of its opcode, operands and its format.

Specification of the rISA model in EXPRESSION consists of describing the rISA instructions and the restrictions on the operands. Each rISA instruction is specified as the compressed counterpart of a normal instruction. The register accessibility restrictions are specified by considering the rISA registers as a special class of registers (that is a subset of the general purpose register class). Furthermore, the limitations on the immediate values that can be specified in a rISA instruction are also specified. Finally, some special rISA instructions such as the *mx, rISA_mx, rISA_nop, rISA_extend, rISA_load, rISA_store* and *rISA_move* are identified in the specification. This information in the EXPRESSION description is used to derive a rISA architecture model (Figure 3.13) that is used by the retargetable compiler and the simulator.

3.7.4 Compiler-in-the-loop DSE

Figure 3.14 shows the flow of the EXPRESSION retargetable compiler. The compiler takes the rISA model described in EXPRESSION as input, and is able to retarget itself to generate good quality code for the machine with rISA model described in EXPRESSION. The EXPRESSION description is also used to generate the simulator for the machine. Using the rISA instructions to normal instructions mapping, the translator unit is generated, and is pre-appended to the decode unit. By considering the compiler effects during DSE, the designer is able to accurately estimate the impact of the various rISA features.

3.7.5 rISA Model

In this section, we briefly describe the rISA processor model. The model defines the rISA IS, and mapping of rISA instructions to normal instructions. A rISA instruction

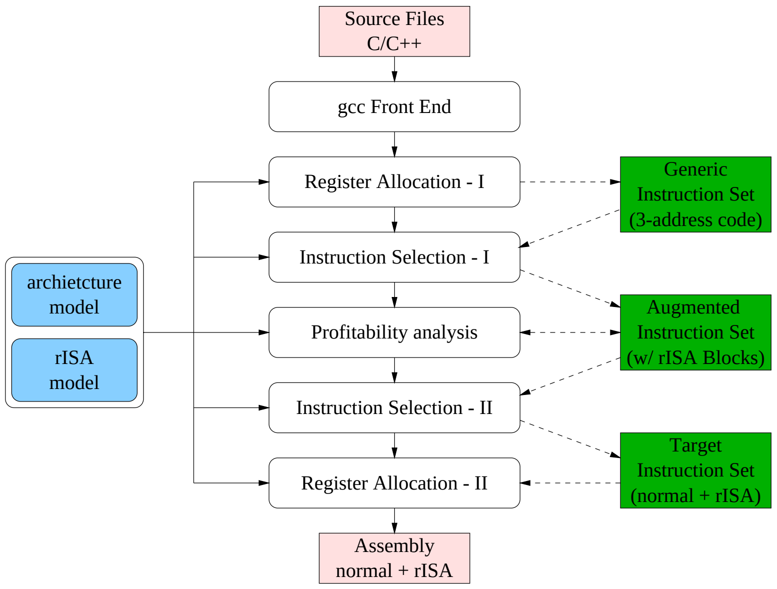


Figure 3.14: rISA Compiler flow for DSE

should map to a unique normal instruction. Such a mapping simplifies the translator unit, so that it does not cause any performance delay.

As rISA processors can operate in either the rISA mode or in the normal mode, a mechanism to specify the mode is necessary. For most rISA processors, this is accomplished using explicit instructions that change the mode of execution. We term an instruction in the normal IS that changes mode from normal to rISA the *mx* instruction, and an instruction in the rISA IS that changes mode from rISA to normal the *rISA_mx* instruction.

Every sequence of rISA instructions starts with an *mx* instruction and ends in a *rISA_mx* instruction. To ensure that the ensuing normal instruction aligns to the word boundary, a padding *rISA_nop* instruction is needed.

Due to bit-width constraints, a rISA instruction can access only a subset of registers. The register accessibility of each register operand must be present in the rISA model. The width of immediate fields must also be specified.

In addition, there may be special instructions in the rISA model to help the compiler generate better code. A very useful technique to increase the number of registers accessible in rISA mode is to implement a *rISA_move* instruction that can access all registers (This is possible because a move has only two operands and hence has more bits to address each operand.). A technique to increase the size of the immediate value operand is to implement a *rISA_extend* instruction that completes the immediate field of the succeeding instruction.

Numerous such techniques can be explored to increase the efficacy of rISA architectures. In the next section we describe some of the more important rISA design parameters that can be explored using our framework.

3.7.6 rISA Design Exploration

Due to highly constrained design of rISA, the code compression achieved is very sensitive to the rISA chosen. rISA design space is huge and several instruction set idiosyncrasies make it very tough to characterize. To show the variation of code compression achieved with rISA, we take a designer's approach. We systematically design several rISA, and study the code compression achieved by them. We start with the extreme rISA designs of *rISA_7333* and *rISA_4444* and gradually improve upon them.

The first rISA design point is (*rISA_7333*). In this rISA, the operand is represented by 7-bits, and each operand is encoded in 3-bits. Thus there can be 2^7 instructions in this rISA, but each instruction can have access to only 8 registers, or be a constant that can be represented in 3-bits. However, instructions that have 2 operands (like move) have 5-bit operands, thus they can access 32 (= all the registers in our architecture model) registers. Owing to the uniformity in the instruction format, the translation unit is very simple for this rISA design.

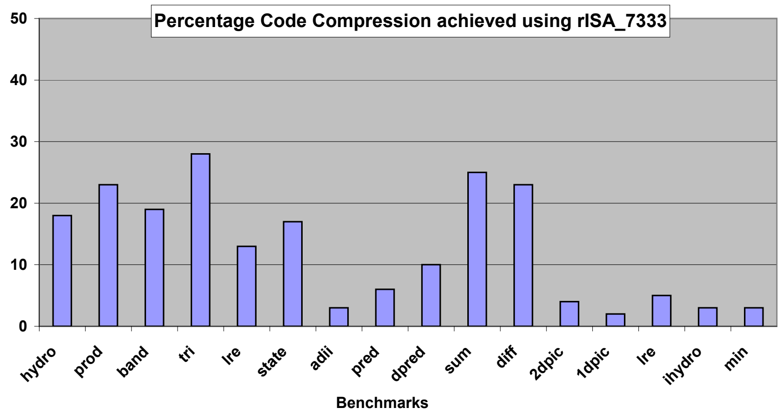


Figure 3.15: Percentage code compression achieved by using rISA_7333

Figure 3.15 shows that the *rISA_7333* design on an average achieves 12% code compression. EXPRESS is unable to achieve good code compressions for applications that have high register pressure, e.g., *adii*, and those with large immediate values. In such cases, the compiler heuristic decides not to rISAize large portions of the application to avoid code size increase due to extra spill/reload and immediate extend instructions.

The first limitation of *rISA_7333* is overcome in the second rISA design i.e. *rISA_4444*, which takes us to the other limit. In this rISA the opcode as well as each operand is encoded in 4-bits. Although only 16 instructions are allowed in such a rISA design, it allows each operand to access 16 registers. We profiled the applications and incorporated the 16 most frequently occurring instructions in this rISA.

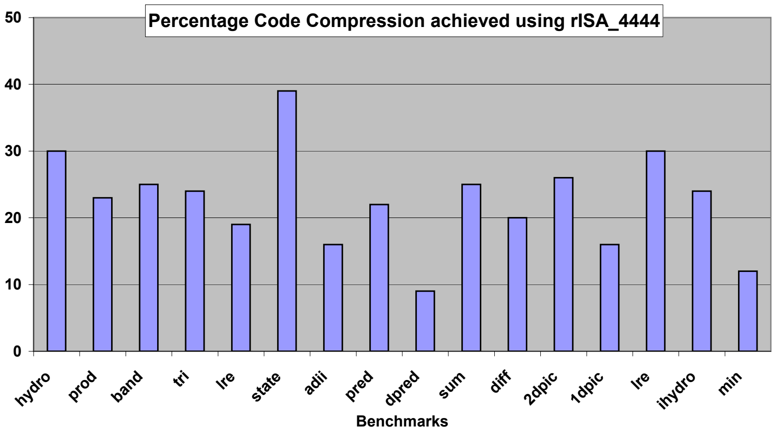


Figure 3.16: Percentage code compression achieved by using rISA_4444

Figure 3.16 shows that the register pressure problem is mitigated in the *rISA_4444* design. It achieves better code size reduction for benchmarks that have high register pressure, but performs badly on some of the benchmarks because of its inability to convert all the normal instructions into rISA instructions. *rISA_4444* achieves about 22% improvement over normal instruction set.

We now attack the second problem faced in *rISA_7333* (small immediate values). For instructions that have immediate values, we decrease the size of opcode, and use the bits to accommodate as large an immediate value as possible. This design point is called *rISA_7333_imm*. Because of the non-uniformity in the size of the opcode field, the translation logic is complex for such a rISA design.

As Figure 3.17 shows the *rISA_7333_imm* design achieves slightly better code compressions as compared to the first design point since it has large immediate fields, while having access to the same set of registers. *rISA_7333_imm* achieves about 14% improvement over normal instruction set.

Another optimization that can be performed to save precious bit-space is to encode

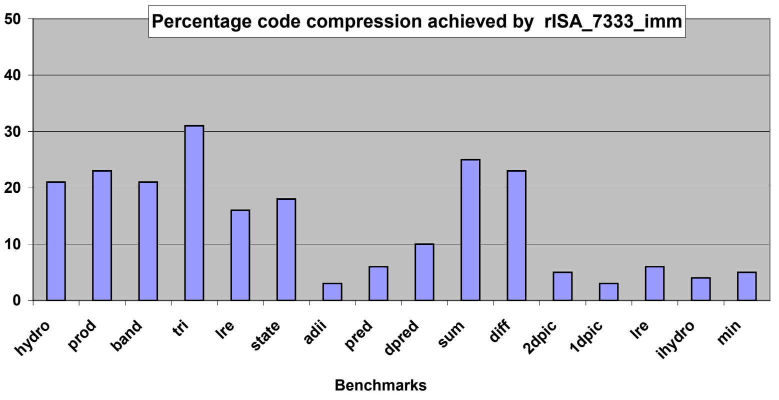


Figure 3.17: Percentage code compression achieved by using rISA_7333_imm

instructions with same operands with different opcode. Since fewer operands are required to express such an instruction, it requires lesser instruction bits, which can be used in two ways, first is the direct way by providing increased register file access to the remaining operands and second (a more indirect way) is that this instruction can afford a longer opcode, another instruction which has tighter constraints on the opcode field (example an instruction with immediate operands) can switch opcode with this instruction. We apply the implied operands feature in *rISA_7333* and obtain our forth rISA design i.e. *rISA_7333_imp_opnd*. This rISA design matches the MIPS16 rISA.

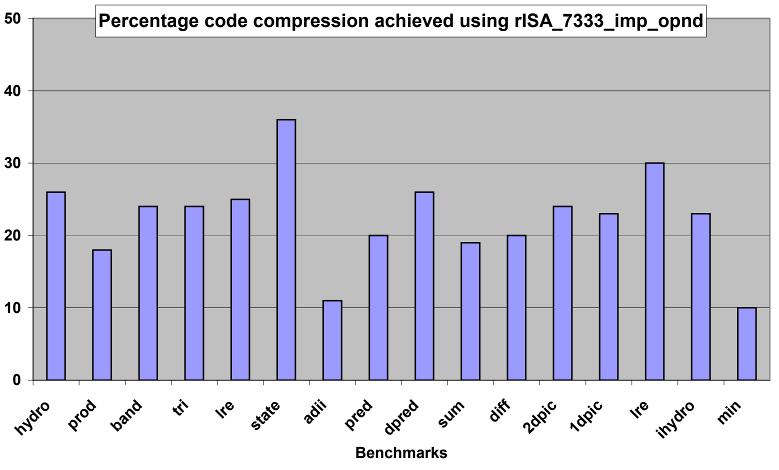


Figure 3.18: Percentage code compression achieved by using rISA_7333_imp_opnd

The *rISA_7333_imp_opnd* design achieves, on average, about the same code size improvement as the *rISA_4444* design point. Note that the performance benefits of using implicit operands is substantial for some applications such as *state* and *dpred*. *rISA_7333_imp_opnd* achieves about 22% improvement over normal instruction set.

The fifth rISA design point i.e. *rISA_hybrid* is a custom ISA for each benchmark. All the previous techniques are used to define a custom rISA for each benchmark. In this rISA design instructions can have variable register accessibility. Complex instructions with operands having different register set accessibility are also supported. The register set accessible by operands varies from 4 to 32 registers. We profiled the applications and manually (heuristically) determine the combinations of operand bit-width sizes that provide best code size reduction. The immediate field is also customized to gain best code size reduction.

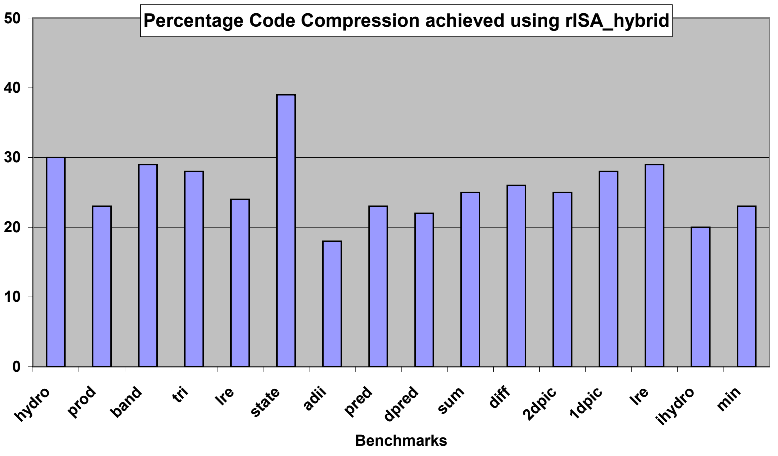


Figure 3.19: Percentage code compression achieved by using rISA_hybrid

The *rISA_hybrid*, because it is customized for the application set, achieves the best code size reduction. *rISA_Hybrid* achieves about 26% overall improvement over normal instruction set. The code compression is consistent across all benchmarks.

Figure 3.20 shows the average (over benchmarks) reduction in code compression achieved in various rISA designs. It can be deduced from the experimental results presented that the code compression achieved is very sensitive to the application characteristics and the rISA itself. Choosing the correct rISA for the applications can result in up to $26\% - 12\% = 14\%$ more code compression. Thus it is very important to design and tune the rISA to the applications.

We have shown that the register pressure heuristic consistently achieves high code compressions (up to 22%). We also observe that the code compression obtained is very sensitive on the application and the rISA itself. Therefore there is a need to effectively

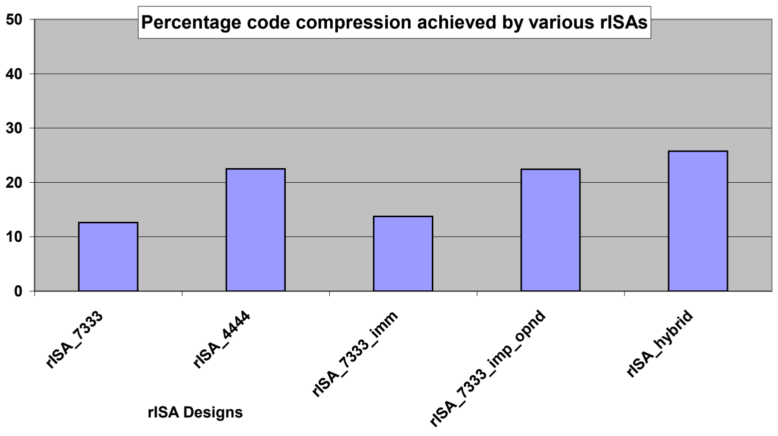


Figure 3.20: Code size reduction for various rISA architectures

explore the rISA design space and that high degrees of code compression can be achieved by tuning the rISA for specific applications.

3.8 Summary

The instruction set of a processor defines how the software can utilize the internals of the processor through instructions. Furthermore it is a primary mechanism for a compiler to optimize for power and performance of the processor. Consequently the instruction set of the processor is very carefully designed, and there has been a lot of research in developing compiler techniques to exploit the features of the instruction set of a processor for power and performance improvements.

In this chapter, we focused on a popular instruction set feature called *reduced bit-width Instruction Set Architecture* (rISA). Architectures with rISA have two instruction sets, one set comprises the *normal* 32-bit wide instructions, while the other is composed of *narrow* 16-bit wide instructions. Most popular embedded processors, e.g., ARM, MIPS implement rISA. The intuition behind such "dual" instruction set architectures is the observation that in RISC systems, very few instructions are used majority of the time. Thus if the most frequently used instructions could be compressed in 16-bits, then up to 50% code size reduction can be achieved by representing the application in terms of instructions of the narrow instruction set.

The narrow instruction set has limited expressibility: not all normal instructions can be converted into narrow instructions, and some normal instructions translate into multiple narrow instructions. In addition, narrow instructions have access to only a few registers.

Existing compilation techniques perform the conversion from normal instructions to narrow instructions at a routine level of granularity. Since all normal instructions do not have a corresponding narrow instruction, very often the routine cannot be compressed at all. Even if it is converted, due to limited register accessibility of registers from the narrow instructions, the register pressure in the narrow instruction region increases, resulting in spill code and therefore an increase in code size. Existing techniques are not aware of this code size increase due to register pressure. Owing to these two reasons, existing code compression techniques using rISA are therefore able to achieve only 14% code compression.

We proposed a novel register pressure sensitive compilation technique for rISA architectures. Further we perform this conversion at instruction level of granularity, increasing the scope of profitable conversion to narrow instruction set and therefore higher code compressions. Our approach is able to consistently achieve 35% code compression.

Further, we find that the code compression is very sensitive on the compiler and the rISA instruction set. Therefore it is very important to include the compiler effects during the design of rISA instruction set. Consequently we proposed a CIL DSE framework for exploring the rISA design space. CIL DSE demonstrates that correctly choosing the rISA design can result in upto 2X improvement in achievable code compression.

Due to codesize reduction using rISA, the front end of the processor has to fetch fewer number of bytes from the instruction cache. Using rISA architectures therefore also helps in reducing the instruction cache energy consumption. Although most previous research has focused on evaluating this energy reduction, we also developed a compilation technique for rISA architectures aimed at reducing the processor energy consumption [32]. Thus the rISA ISA architectural feature is a very useful technique for code size reduction and processor power reduction for both general purpose purpose and embedded processors. We believe this feature is especially pertinent for embedded processors, where design constraints (e.g., power, code size etc.) other than performance can be equally or even more important.

Chapter 4

Processor Pipeline - Memory Interface Design

The last few decades have seen dramatic improvements in microprocessor performance. This has been largely due to technological improvements and architectural innovations. In contrast, the DRAM (Dynamic RAM) performance has not improved at a commensurate pace, leading to the infamous ”processor-memory gap”. Memory chip manufacturers have tried to bypass the weak performance of the memory core by designing complex memory interfaces which isolate the behavior of the slow memory core from the fast interconnections between the memory chip, the memory controller, and the processor pipeline. In most modern processors, this interface is implemented using a bus-based system.

High-end systems employ large size and high associativity caches, which help in hiding the latencies of slow memories. However, embedded systems may not have caches, or may only have very small caches. Consequently, the memory latency has a direct impact on the power and performance of embedded processors. Thus the processor pipeline - memory interface design, as depicted in Figure 4.1 is a very crucial step of embedded processor design.

Software prefetching [41] has been a primary mechanism for hiding memory latency. Prefetching attempts to utilize free cycles of the processor pipeline - memory interface to get data that will be required by the processor in the future; thus improving the efficiency of the processor pipeline - memory interface. Prefetching in software is effective and preferred because of two main reasons. First is the extremely low overhead in the hardware, and second is that software prefetching tends to be more accurate and therefore has low run-

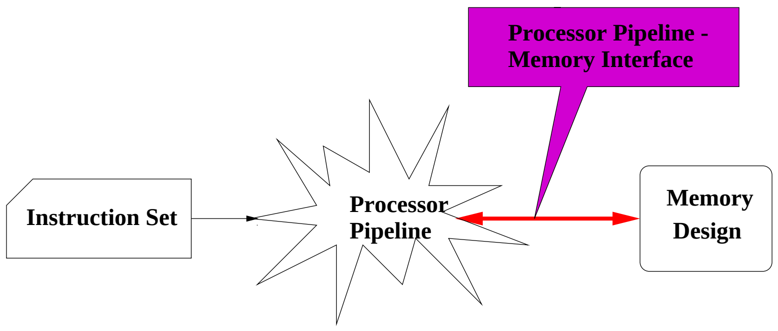


Figure 4.1: Processor Memory Interface Design Abstraction

time overhead due to cache pollution. Although software prefetching can dramatically improve the power and performance of embedded processors, most previous research has focused on using software prefetching to improve the performance of the processor, and obtain power reductions only as a byproduct.

In this chapter we develop novel technique that uses software prefetching to reduce the processor power consumption by exploiting the knowledge about the processor pipeline - memory interface. We then demonstrate that pareto-optimal design points can be discovered by CIL DSE of processor pipeline - memory interface design.

4.1 Processor-Memory Bus

Memory customization is one of the most important steps in embedded processor design because of its significant impact on the system performance, cost and power consumption. Memory characteristics (like latency, bandwidth, etc.) are numerous and complex, but they should be matched carefully to the application requirements and the processor computation speed. On one hand, processor stalls (processor waiting for data from memory) should be minimized, while on the other hand, idle memory cycles (memory waiting for requests from processor) should be reduced. A lot of time and effort of experienced designers is invested in tuning the memory characteristics of embedded systems to target application behavior and processor computation.

However even after such careful design of the memory system there are times during the execution of real-life applications when the processor is stalled, and there are times when the memory is idle. We define a *processor free stretch* to be a sequence of contiguous cycles when the processor is stalled. Figure 4.2 plots the lengths of processor free stretches

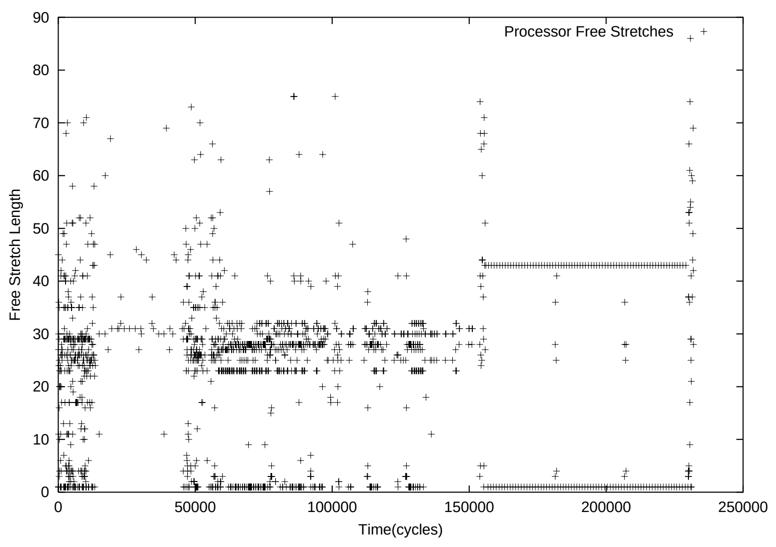


Figure 4.2: Length of Processor free stretches

over the execution of the *qsort* application from the MiBench benchmark suite[20] running on the Intel XScale [19]. Very small processor free stretches (1-2 cycles) represent processor stalls due to pipeline hazards (e.g., Read After Write hazard). A lot of free stretches are approximately 30 cycles in length. This reflects the fact that the memory latency is about 30 cycles. An important observation from this graph is that although the processor is stalled for a considerable time (approximately 30% of the total program execution time) the length of each processor free stretch is small. The average length of a processor free stretch is 4 cycles, and all of them are less than 100 cycles.

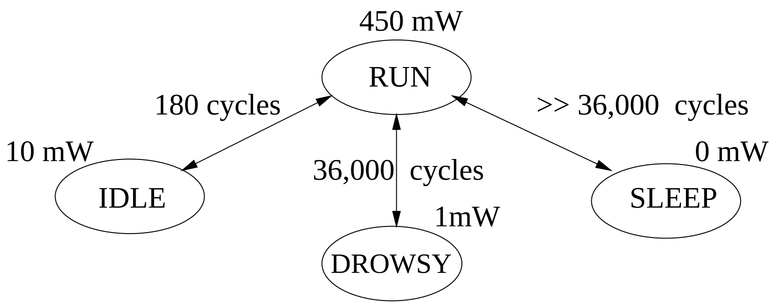


Figure 4.3: Power State Machine of XScale

A processor free stretch is an opportunity for optimization. System throughput and energy can be improved by temporarily switching to a different thread of execution [42]. The energy consumption of the system may be reduced by switching the processor to a low power state. Figure 4.3 shows the power state machine [43] of the Intel XScale. In the default mode of operation, the XScale is in RUN state, consuming 450mW. XScale has three low power states: IDLE, DROWSY, and SLEEP. In the IDLE state the clock to the processor is gated, and the processor consumes only 10mW. However, it takes 180 processor cycles to switch to and back from the IDLE state. In the DROWSY state the clock generation circuit is turned off, while in the SLEEP state the processor is shut down. DROWSY and SLEEP states have much reduced power consumption, but at an increased cost of state transition. The break-even processor free stretch for a transition to IDLE state is $180 \times 2 = 360$ cycles. This implies that the processor free stretch should be more than 360 cycles to profitably switch the processor to IDLE state. Clearly existing processor free stretches are not large enough to achieve power savings by switching to a low-power state. It is important to note that this is in-spite of the fact that the total processor stall duration is quite large.

The traditional assumption has been that power optimization opportunities by switching the processor to low-power state can only be found at the operating system level (inter-process), and not at the compiler-level (intra-process). Consequently a lot of research has been done to develop power optimization techniques, that operate application level granularity, and are controlled by the operating system. However, to the best of our knowledge there has been no research to aggregate processor free times within an application to reduce the power/energy consumption of the processor.

4.2 Related Work

Prefetching can be thought of as a technique to aggregate memory activity. [41] presents an exhaustive survey of hardware, software, and cooperative prefetching mechanisms. Processor free cycles can be used to increase the throughput of systems by switching to a different thread of execution [42]. They can also be used to reduce the power consumption of a processor by switching it to a low-power mode [44]. Clock gating, power gating, frequency scaling and voltage scaling provide architectural support for such low-power states [44].

Several power/energy reduction techniques work at the operating system level by switching the whole processor to a low-power mode. The operating system estimates the processor free time using predictive, adaptive or stochastic techniques [43]. Some power/energy reduction techniques operate at the application level but they control only a very small part of the processor, e.g. multiplier, floating point unit. The compiler statically or hardware dynamically estimates the inactive parts of processor, and switches them to low-power mode [45].

However, there are no existing techniques to switch large parts of processor to a low-power state, during the execution of an application. This is mainly because during the execution of an application, without aggregation the processor rest times are too small to profitably make the transition to the low-power state. This is true even though the total stall duration may be quite significant. In this chapter we present a technique to aggregate small processor stalls to create a large free chunk of time when the processor is idle, and can be profitably switched to a low-power state.

4.3 Motivation

Consider the loop expressed in Figure 4.4(a). In each iteration the next element from two arrays *a* and *b* are loaded and their sum is stored into the third array *c*. The ARM assembly code of the loop generated by GCC is shown in Figure 4.4(b).

```
for (i=0; i<1000; i++)
   c[i] = a[i] + b[i];
```

(a)

```
1. L: mov   ip, r1, lsl#2
2.    ldr   r2, [r4, ip]
3.    ldr   r3, [r5, ip]
4.    add   r1, r1, #1
5.    cmp   r1, r0
6.    add   r3, r3, r2
7.    str   r3, [r6, ip]
8.    ble   L
```

(b)

Figure 4.4: Example Loop

For our processor pipeline - memory interface experiments, we employ a very simple processor pipeline and memory model, but a complex memory interface model. Our processor pipeline, memory and the interface model is described in Figure 4.5. In our simple processor pipeline model, every instruction takes one cycle to execute, if the required data is in the cache. In our simple memory model, memory can service a request in 6 processor cycles. To model a realistic interface behavior we need a more complex model. We assume there is a request buffer in front of the cache. This makes the cache non-blocking (the processor does not stall on a cache miss). The request latency of the memory (time elapsed between making a request to getting the first word) is 12 cycles. The memory bus is pipelined so that multiple requests can be pending and hide the memory latency. We assume independent request and data bus for increased efficiency of memory bus. The memory bandwidth is 1 word per 3 cycles and the cache line size is 16 bytes.

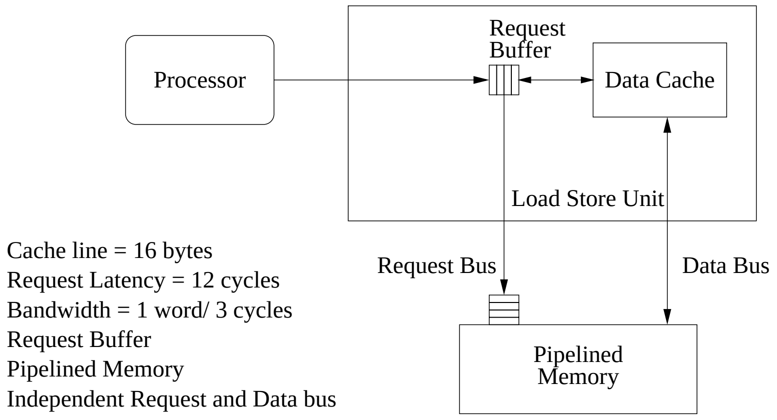


Figure 4.5: Example Memory Architecture

To simplify the analysis we assume simple loops with sequential array accesses, like the loop in Figure 4.4. The analysis can be easily generalized later. For such loops it is convenient to think of memory operations per k loop iterations; where k is the number of elements (of the type that are being accessed in the loop) that fit in one cache line. For example the loop in Figure 4.4 loads integers(4 bytes), thus $k = 16/4 = 4$. Therefore we define an ITER as 4 iterations. In one Iteration, the loop needs to load 3 lines, one line each from arrays a, b, and c.

The Computation C in a loop is defined to be the number of cycles required to execute one ITER of the loop if there are no cache misses. For this loop, $C = 8 \times 4 = 32$ cycles. The Memory Latency ML of a loop is defined as the number of cycles required to transfer all the data required in one ITER in steady state, assuming that the request was made well in advance. In every ITER in steady state, three lines have to be loaded (one line each from arrays a, b, and c) and one line has to be written back (one out of every three lines, *the one corresponding to array c*, is dirty). Thus $ML = 4 \times 4 \times 3 = 48$ cycles.

For this loop $ML > C$. We call such a loop as memory bound, and there is no way to avoid processor stalls for such a loop. Even if the request for the data is made well in advance, the memory bandwidth is not enough to transfer the required data in time. For loops in which $ML = C$, the memory requirement and computation are matched. The memory system can be said to be tuned for such loops.

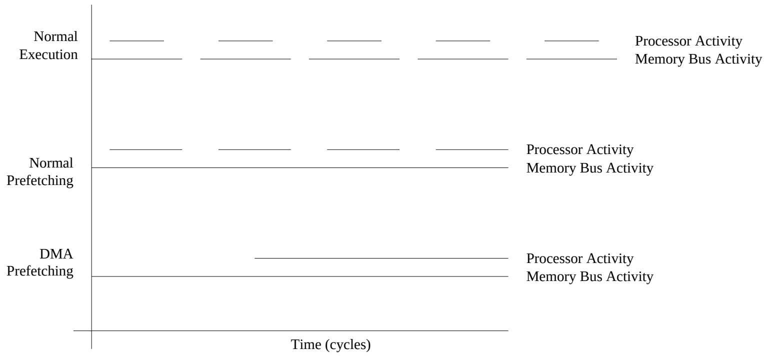


Figure 4.6: Processor Memory Activity

The top two sets of lines in Figure 4.6 represent the processor and memory bus activity in a normal processor. The processor executes intermittently. It has to stall at the 6^{th} instruction of each ITER, and wait for the memory to catch up. The memory activity starts with the request made by the 2^{nd} instruction, but data continues to transfer on the memory bus, even after the processor has stalled. Even before all this data could be transferred, there is a new request from the 2^{nd} instruction of the next ITER of the loop, and the request latency could not be completely hidden. Thus in a normal processor, both the processor and memory bus activities are intermittent.

Prefetching has the effect of aggregating the memory bus activity. The next two horizontal lines from the top in Figure 4.6 represent the processor and memory bus activity with prefetching. Each period of processor execution is longer because of the computational overhead of prefetching. However, the memory bus activity is now continuous. The memory bus is one of the most critical resources in several systems, including our example architecture. Improving the utilization of such an important resource is very important. As a result of this, the runtime of the loop decreases.

However, the processor utilization is still intermittent. If we can aggregate the processor utilization, we can achieve an activity graph as shown by the bottom two lines of Figure 4.6. The plot clearly shows that the processor needs to be active only for a fraction of time of the memory bus activity, the fraction being equal to C/ML. Furthermore the aggregated inactivity window can be large enough to enable profitable switching to low-power mode, and energy reduction achieved. In this chapter, we present a code transformation and architectural support required to achieve this aggregation.

4.4 Our Aggregation Approach

```
for (i=0; i<1000; i++)
   c[i] = a[i] + b[i];
```

(a)

```
startPrefetch()
for (i1=0; i1<T2; i1+=T)
   setProcWakeup(w)
   procIdleMode()
   for (i2=i1; i2<i1+T; i2++)
      c[i2] = a[i2] + b[i2]
```

(b)

Figure 4.7: Loop Transformation

Our idea of aggregating processor rest time is very intuitive, and requires a prefetch engine. Figure 4.7 shows the transformation of the loop shown in Figure 4.4. For each memory bound loop in the application we find out which arrays and how many lines from each array are required by the loop and the prefetch is started. The loop is first tiled with tile of size T. Inside each tile, the wakeup factor w is set in the prefetch engine to wake up the processor. The processor is then switched to a low-power state. In the low-power state, the processor waits for an interrupt from the prefetch engine. All parts of processor except the load store unit are switched to a low-power state. At appropriate time the prefetch engine generates an interrupt to wake up the processor. The processor wakes up and resumes execution on the prefetched data in the cache. The prefetch engine continues to work even after the processor is awake. The tile size T, and the wake up parameter w are computed so as to maximize the time the processor is in the low-power state.

Note that Figure 4.7 shows only the transformation of the loop kernel. However to implement this transformation, we need to identify the loop kernel, the epilogue and the prologue of the loop. First we describe the complete loop transformation steps, then we show the calculation of T and w to achieve the maximum energy savings, and finally describe our prefetch engine implementation.

1. Identify simple memory bound loops a. C can be estimated by assembly code of loop b. ML can be estimated by source code of loop
2. Break the loop into two parts a. Before steady state b. At steady state
3. Find MLof the first part of loop by profiling Compute wakeup time for first part of loop
4. Find tiling factor and wakeup time for second part of the loop Break away the third part (tail) of the loop
5. Find the wakeup time for the third part of loop

Figure 4.8: Transformation Steps

4.4.1 Code Transformation

The transformation is applied only to memory bound loops. The loop transformation described in Figure 4.8, converts the loop in Figure 4.4 into the loop in Figure 4.9. The first step in the transformation is to estimate the memory latency ML and the computation C of the loop. The ML can be estimated by a simple source code analysis, while C can be estimated by a simple assembly code analysis. We define the steady state of the loop, as when the loop has consumed a cache-full of data. Before steady state, the memory latency of the loop may not be equal to ML. If the data required in an ITER is already present in the cache, there will be no memory transfers. On the other hand, if the data residing in the cache is dirty, then more writebacks will happen. The loop is therefore broken into two parts, the first one is before steady state ($i = 0; i < T1$), the prologue, and the second one is at steady state ($i = T1; i < N$). For this we need to estimate $T1$, the break point of the loop.

Before steady state, profile information is needed to estimate ML. A cycle accurate simulator is instrumented to count all memory activity in the loop in the first part of the loop. Thus for the prologue, $ML = \mathit{cache\_misses}/T1$. Using ML, C and $T1$, the wake up parameter $w1$ for the first part of the loop is computed.

For the second and the main part of the loop, ML and C are known. The tile size T of the loop needs to be computed. There will be $(N - T1)/T$ tiles in the second part of the loop. However, there may be a tail end of the loop left ($i = ((N - T1)/T) \times T; i < N$).

The tail end of the loop, is separated as the third part or epilogue of the loop. In the main kernel, the processor will be switched to low-power mode at the beginning of each tile, and processor wake up factor will be setup in the prefetch engine. The processor wakes up by the interrupt from the prefetch engine, and starts executing. The wake up factor w is computed in such a way that the tile computation ends by the time processor exhausts the prefetched data. The wake up time $w2$ for the epilogue is also estimated.

```
// Set the prefetch engine
1. setPrefetchArray a, N/L
2. setPrefetchArray b, N/L
3. setPrefetchArray b, N/L
4. startPrefetch

// first part of the loop
5. setProcWakeup w1
6. procIdleMode
7. for (i1=0; i1<T1; i1++)
8.    c[i1] = a[i1] + b[i1]

// tile the second part of the loop
9. for (i1=0; i1<T2; i1+=T)
10.   setProcWakeup w
11.   procIdleMode
12.   for (i2=i1; i2<i1+T; i2++)
13.      c[i2] = a[i2] + b[i2]

// tail of the loop
14. setProcWakeup w2
15. procIdleMode
16. for (i1=T2; i1<N; i1++)
17.    c[i1] = a[i1] + b[i1]
```

Figure 4.9: Fully Transformed Loop

Currently our analysis of aggregating processor free times is for loops that request consecutive lines from each array. It should be possible to extend the scope of this transformation by employing more sophisticated data analysis techniques. The overhead of the additional instructions inserted by our transformation is negligible ($<1\%$ of the total runtime).

4.4.2 Computation of Loop breakpoints, Tile size and Wake up Factor

The loop breakpoints, tile size, and wake up time depend on the the loop characteristics, cache parameters, and the location of arrays in memory. For our analysis we assume that C is the computation, and ML is the memory latency of the loop. Also assume that in each ITER of the loop, r lines need to be read. Further assume that the cache has s sets, and has associativity a.

We first compute the first breakpoint $T1$ of the loop. The steady state starts when the loop has consumed one cache-full of data. In s ITERs, the loop completely uses r lines of each set. Thus it will use the whole cache and a little more in $T1 = a \times s/r + s$ ITERs. To compute the wake up time $w1$ of the processor, given the number of ITERs the loop will execute, the following condition must hold true, $C \times T1 = ML \times (T1 - w1/r)$. This ensures that enough data has been loaded into the cache to perform computation without any cache misses. Thus $w1 = ((ML - C) \times T1 \times r)/ML$.

The wake up time w of the processor inside the tile is computed such that each time we prefetch a cache-full of data, thus $w/r = (a/r) \times s$. The tile size T can then be computed by the following relation, $(C \times T/ML) + w/r = T$. This implies $T = w/r \times ML/(ML - C)$.

The wake up time $w2$ for the third part of the loop can be computed similarly as $w2/r = ((ML - C) \times (N - T2))/ML$.

4.4.3 Architectural Support

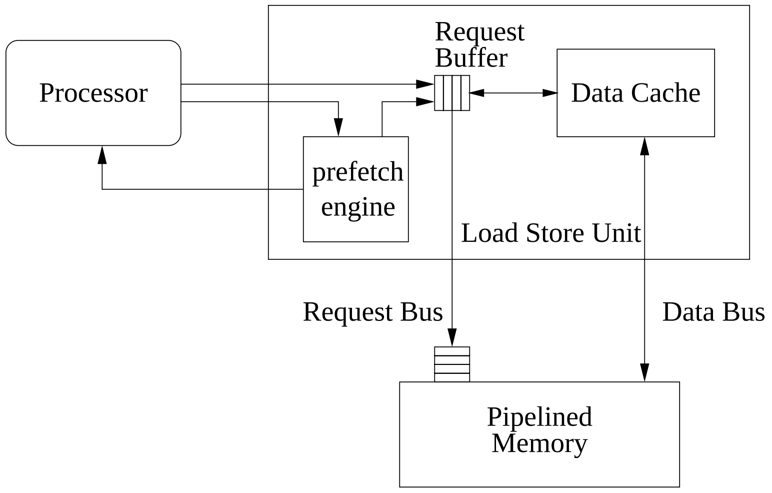


Figure 4.10: Prefetch Engine

A minimal amount of architectural support is needed for the proposed scheme to work. The exact implementation of the prefetch engine as well as its interface may differ for each architecture. Our implementation is shown in Figure 4.10. A mechanism to set up the prefetch engine is required. The processor needs to specify the start of the arrays, and the number of lines of each array to be prefetched to the prefetch engine. The processor also needs to specify the number of lines to prefetch before waking up the processor (w). A mechanism to start the prefetch engine and a way to switch the processor to a low-power mode is needed. In the low power mode, the clock to the processor(other than the load store unit) can be frozen, and even powered-down. The processor just waits for an interrupt from the prefetch engine. The prefetch engine keeps prefetching by inserting requests in the request buffer. The prefetch engine monitors the request buffer, and it does not let the request buffer be empty. If the request buffer is always full, the activity on the data bus remains uninterrupted. This ensures the maximum usage of data bus by keeping the request buffer non-empty. After the prefetch engine has requested more than w lines from the memory, it should generate an interrupt to wake up the processor. The processor should catch the interrupt and resume execution again. The prefetch engine continues its operation even after waking up the processor. After all the data has been prefetched, the prefetch engine should disengage itself, until it is invoked by the processor again.

4.5 Effectiveness of Aggregation

To demonstrate the usefulness and efficacy of our approach we perform experiments on a XScale simulator that we developed. We validated our simulator against the 80200 EVB (XScale Evaluation Board) [21] to be accurate within 7% on an average in cycle count measurements. The simulator has been extended to model the prefetch engine. The code transformations have been applied on the best performing code generated by GCC.

4.5.1 Steady-state Analysis

We perform the first set of experiments on kernels from the Stream suite[46]. Stream kernels are widely used to balance the memory bandwidth to the computation speed of processors. The kernels have varying degree of computation to memory requirement (C/ML) ratio.

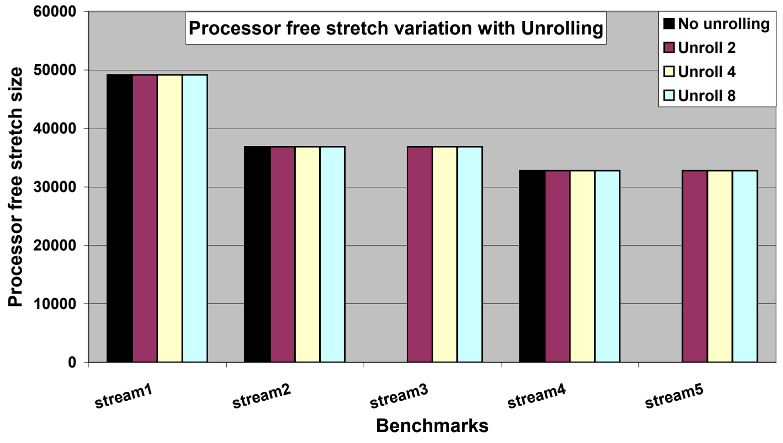


Figure 4.11: Variation of processor free stretch size with Unrolling

The leftmost (black) bars in Figure 4.11 plot the processor free stretch our aggregation technique could obtain for each kernel in Stream suite. Up to 50,000 processor free cycles can be accumulated. Two of the kernels, *stream3* and *stream5*, do not have black bars. Our technique was unable to aggregate processor free stretches for these two kernels, because $C > ML$ for these kernels.

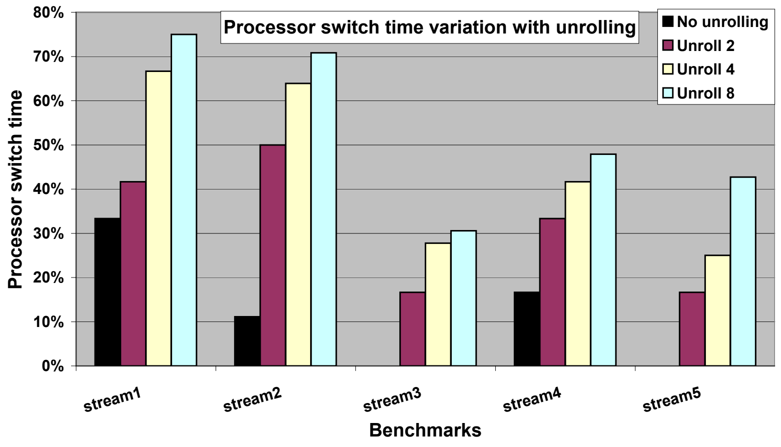


Figure 4.12: Variation of processor switch time with Unrolling

The leftmost (black) bars in Figure 4.12 represent the percentage of total execution time, the processor can be switched to a low-power mode (processor switch time). The processor state switching time (360 cycles for each switch), is considered to be full-power mode. The processor can be switched to a low-power mode for up to 33% of the program execution time. Note again, that the processor cannot be switched to a low-power mode in *stream3* and *stream5*. This shows that our technique is able to aggregate processor free times into large chunks, so that it becomes profitable to switch the processor to a low-power mode. Furthermore, our results show that the processor can be switched to low-power mode for a significant amount of kernel run-time.

4.5.2 Effect of Loop Unrolling

The processor free time our technique is able to aggregate is proportional to the ratio, C/ML. Optimizations that improve this ratio improve the aggregation results. Loop unrolling is a popular optimization that reduces the computation per ITER (C) of a loop. In this subsection we analyze the impact of loop unrolling on the processor free stretch and the processor switch time obtained. Figure 4.11 shows that the processor free stretch size obtained is not affected by unrolling. As previously discussed, the ability to accumulate processor free time is dependent on whether $ML > C$, but if it is possible, the size of the processor free stretch obtained is proportional to ML. It does not increase by reducing C. Figure 4.12 show plots the processor switch time percentage obtained for the Stream kernels for different unroll factors. The first observation is that after unrolling the processor can find switching opportunities even for kernels *stream3* and *stream5*, which were earlier deemed unprofitable. The second important observation is that unrolling increases the processor switch time percentage for the kernels. The processor can now be switched to a low-power mode for up to 75% of kernel run-time. Thus unrolling improves the applicability and efficacy of our aggregation technique. It is worth mentioning here that hand-generated assembly code has even better C/ML ratio.

Another interesting thing to note here is that hand generated assembly typically is much denser, i.e., it uses much fewer instructions to code the same functionality (thereby reducing C) , but the ML remains the same. Thus the aggregation techniques works even better on hand-generated assembly code. This technique is therefore particularly applicable for a large set of Digital Signal Processing (DSP) applications, where many important kernels are often hand coded. We believe our technique shows significant promise for energy reduction in such designs.

4.5.3 Energy Saving Estimation

In this subsection we estimate the energy consumption of the processor with and without applying our aggregation technique. We develop simple and conservative state-based processor energy models. We show that even with such conservative energy models, our technique achieves significant processor energy reduction. In the normal RUN mode, operating at 600 MHz, the XScale consumes 450 mW. In the STALL mode, the XScale consumes 25% of 450 mW = 112.5 mW [47, 19, 48]. In the aggregated processor free time, we switch the processor to our low-power mode called MY_IDLE. In the normal IDLE mode of XScale, the clock to the processor and memory are gated, and XScale consumes 10 mW. However in our idle state, MY_IDLE, the memory clock needs to be on, the prefetch engine is on, and we need to keep performing writes in the cache. The clock consumes about 20% power in the XScale, i.e., 90 mW. The clock power is divided into processor clock power and memory clock power. We assume that processor clock is 6 times faster than memory clock. The capacitive load on the processor clock is much more than the capacitive load on the memory clock. We conservatively assume the same capacitive load, therefore memory clock consumes 90/7 = 13 mW. Caches in XScale consume about 25% power, i.e., 112.5 mW. However this is divided into data cache power and the instruction cache power. In XScale the instruction cache and data cache have the same architectural parameters. However on an average the instruction cache is accessed every cycle, while the data cache is accessed every 3 cycles. Thus energy consumed by the data cache is 112.5/4 = 28 mW. We synthesized the prefetch engine using design-compiler of Synopsys-2001.10 on a 0.8μ library, and estimated its power consumption using Synopsys power-estimate[23]. The power consumption, scaled to the XScale process technology (0.18μ) is less than 1 mW. Thus in the MY_IDLE state, processor consumes 13 + 28 + 1 = 42 mW. However for our experiments, we make a further conservative estimate, and assume processor power consumption to be 50 mW. We augmented the cycle accurate simulator with the power of RUN state, STALL state and MY_IDLE state. Depending on the current state of the processor, the corresponding state power is added and the total energy consumption computed.

Figure 4.13 shows that even with such a conservative estimate, up to 18% processor energy savings can be obtained by our processor free time aggregation technique on multimedia applications running on XScale.

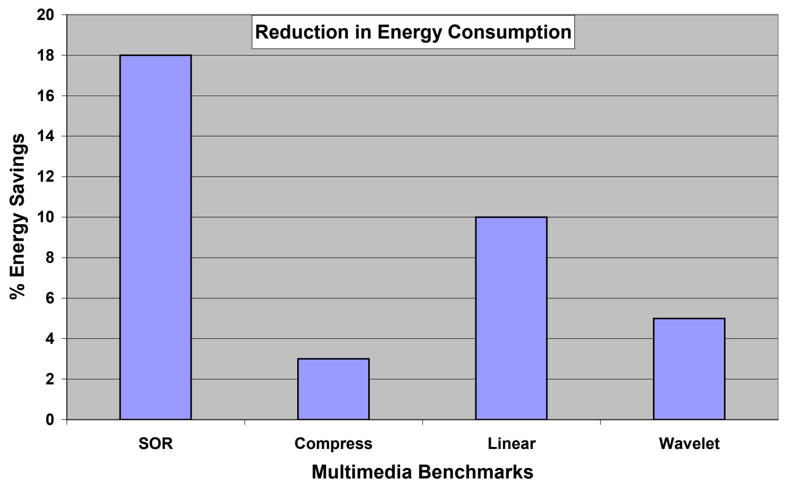


Figure 4.13: Energy savings on multimedia applications.

4.6 Compiler-in-the-Loop Exploration

The increasing gap between the clock rates of the processor pipelines and the memory have resulted in importance of the memory and the memory interface design as one of the most important aspects of processor design.

Although fast memory and high bandwidth memory bus the best for performance, but they may consume too much power to be implemented in embedded processors. Embedded processors for which energy consumption may be even more important than performance, will often slow down the clock of the memory and reduce the bandwidth of the the memory bus to meet the energy budget. Decreasing the memory frequency and bandwidth results in an increase in the time processor is stalled waiting for data from the memory. Since the energy consumed by the processor in stall state is lesser than the busy energy of the processor, the power consumption decreases. The SO DSE will measures the runtime and the power consumption of the processor at each memory bus frequency and bandwidth. The dark diamonds in Figure 4.14 represent the runtime and processor power consumption, while we vary the memory bus frequency and bandwidth.

However, there is further scope of reducing the processor power consumption by switching the processor to low-power IDLE state during the stall periods. But the typical stall durations a normal execution of programs are too small for the processor to profitably switch to low-power IDLE mode. For the Intel XScale embedded processor, running the *qsort* benchmark from the MiBench suite, the average stall duration is 4 processor cycles, with none of the stalls more than 100 processor cycles. On the other hand, it takes 180 processor cycles to switch to low-power IDLE mode. Thus it is not possible to switch to low-power IDLE mode during the memory stalls occurring in traditionally compiled code. However our aggregation technique can aggregate the memory stalls and create a large enough stall so as to profitably switch the processor to the low-power IDLE state. Doing so reduces the processor power consumption. The light squares in Figure 4.14 plot the runtime and processor power consumption, while we vary the memory bus frequency and bandwidth.

In the Figure 4.14 we obtain the runtime and processor power consumption as evaluated by the SO DSE and CIL DSE for each memory bus configuration. The memory bus configuration is defined in terms of two parameters, first is CR, the ratio of the processor clock frequency to the memory bus clock frequency. The second parameter is the width of the memory bus in bytes. Thus if the bus width is b, and clock ratio is c, this configuration implies that b bytes can be transferred between the processor and the memory in c processor clock cycles.

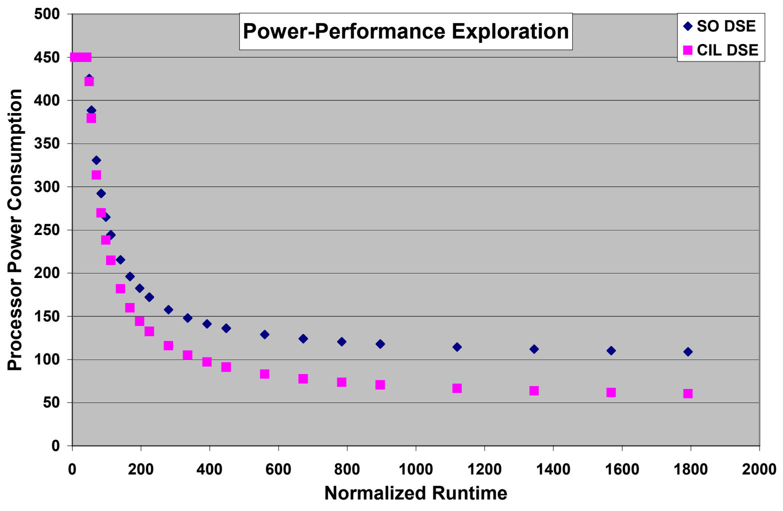


Figure 4.14: Design Space Exploration of Memory Bus Parameters

Figure 4.14 shows that as we traverse the graph from left to right, the memory bus frequency and width decreases, and the runtime increases but the processor power consumption decreases. This is because the processor is now stalled for more time, waiting for data from the memory. For the configurations on the extreme left, the processor and memory is fast enough and there are no processor stalls. Therefore both the SO DSE and CIL

DSE estimate the energy as 450 mW, which is the power consumption of the Intel XScale processor in the busy mode. For all other configurations, the compiler is able to further reduce the processor power consumption by aggregating the processor stalls and switching the processor to low-power IDLE state in the aggregated processor stall duration. As we decrease the memory bus bandwidth, there is more scope for the compiler to reduce processor power consumption. In the limiting case, when the memory is slow enough (extreme right), the SO DSE assumes that the processor will be stalled for most of the time, therefore it estimates the power consumption to be little more than 112.5 mW. 112.5 mW is the power consumption of the processor in stall state. In contrast, CIL switches the processor to low-power IDLE state, and therefore it estimates the processor power consumption to be slightly more than 50 mW. 50 mW is the power consumption of the processor in the low-power IDLE state.

Thus there is a significant difference in the estimation of the processor power consumption between the SO DSE and CIL DSE. SO DSE can severely overestimate the processor power consumption at a given memory bus configuration. Thus it is very important to include compiler effects during the exploration and therefore perform our CIL DSE.

4.7 Summary

The processor pipeline - memory interface may have a huge impact on the effectiveness of the processor as a whole. It is very important to match and tune the processor frequency and the memory frequency, so as to minimize the processor stalling for memory, and reduce the power consumption.

In this chapter, we focused on tuning the bus-based processor pipeline - memory interface. High speed, high bandwidth buses are the best for performance, but they are not only exceedingly costly, but also consume a lot of power. In embedded systems, where power is a major concern, designers implement only slower, low bandwidth buses. A slower, lower bandwidth bus has lower power consumption. In addition the processor is now stalled waiting for results from the memory for longer percentage of time. Since the processor consumes less power when it is stalled then when it is running, the power consumption of the processor also decreases. However there is scope for further processor power reduction by switching the processor to a low-power IDLE state. However memory stalls obtained by executing traditionally compiled codes are not long enough for a profitable switching to the low-power IDLE state. Therefore using existing techniques it is not possible to save processor energy by switching the processor to low-power IDLE mode while stalled for memory response.

Our novel aggregation technique collects memory stall times and creates a large enough stall so that the processor can be switched to the low-power IDLE mode, and thereby save processor power consumption. Our technique is able to aggregate up to 50,000 stall cycles, and by switching the processor to low power IDLE state, the power consumption of the processor can be reduced by up to 18%.

Further, exploring over various memory bus configurations, we find that not considering this achievable lower processor power consumption can result in significantly different evaluation of the processor power consumption. As a result it is very important to consider the compiler effects (power reduction that the compiler technique can reduce) during exploration, and therefore employ CIL DSE techniques.

Although this aggregation technique in its current form may not be applicable to general purpose processor, we believe variants of this technique can be developed for general purpose processors, and we are working in this direction. However, this is already in an extremely important from for embedded processors.

Chapter 5

Memory Design

In most embedded processors, the design of the memory subsystem still remains one of the most important steps of processor design. Significant time and effort of experienced designs is invested in designing the memory subsystem of the processor. A lot of research effort has focused on the design of memory subsystem, which is depicted in Figure 5.1.

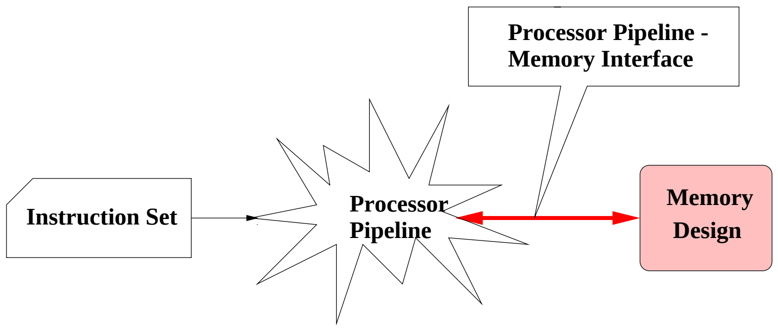


Figure 5.1: Memory Design Abstraction

A broad range of memory designs are popular in embedded processors. Several embedded processors do not use caches at all. In the absence of caches, the memory transfers take a long time. As a result, the role of a compiler is crucial to achieve desired performance by re-ordering the instructions and hiding the memory latency of the system.

However, several high-performance embedded processors employ caches to increase performance. Traditionally the management of caches is hardware controlled: the compiler has no control over it. Caches result in large improvements in performance and power due to the presence of spatial and temporal locality of application data. Previous research has focused on how to optimize the generated code so as to obtain even more benefits from the caches. Compiler techniques such as loop blocking , loop interchange etc. (e.g., [49]) change the order of memory transfers and therefore have a sweeping effect on the power and performance of cache-based processors.

Yet another kind of memory design is based on using scratch pads. Scratch pads are essentially compiler controlled memories. The compiler has to explicitly transfer data to and from scratch pads, and also perform address translation to use them. Due to the reduced overhead, scratch pads are significantly less complex and more power efficient than caches. Several compiler techniques have been developed to automatically map and manage the application data on the scratch pads, e.g., [50] Although compilers may be able to use them very efficiently for regular applications (where the compiler can determine or guess the memory access patterns statically), in general they are difficult to use. Needless to say, the compiler has a huge impact on the effectiveness of such memories.

Thus over the whole range of memory designs, the compiler is instrumental in achieving fast and power-efficient use of the memory subsystem. As a result, the compiler should be involved in the design of the memory subsystem of an embedded processor.

In this chapter we present a novel compilation technique to reduce the memory energy consumption by exploiting a memory architectural feature called Horizontally Partitioned Caches (HPCs). Then we demonstrate the need and usefulness of *Compiler-in-the-Loop* (CIL) Design Space Exploration (DSE) by comparing the memory subsystem energy consumption of the HPC configuration found by CIL DSE against the one found by Simulation-only (SO) DSE.

5.1 Horizontally Partitioned Cache

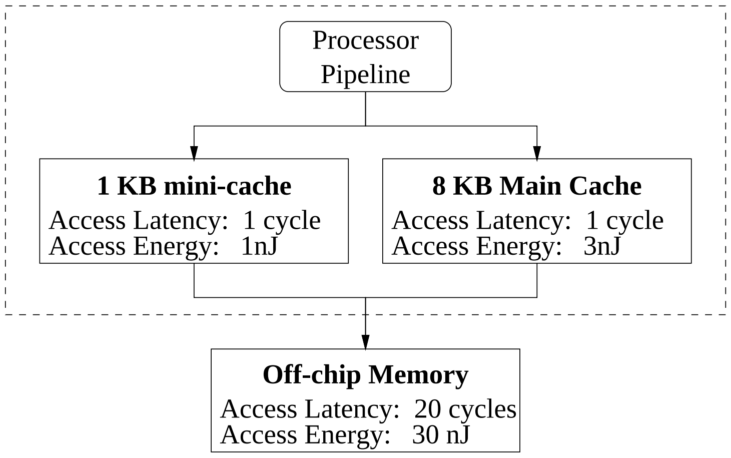
Advances in compiler technology have far from kept pace with the phenomenal pace of microarchitectural innovation. Advanced microarchitectural features are employed in processors to be exploited manually and/or with the hope that the compiler will be able to exploit them in the near future. Horizontally partitioned caches are one such feature. Although originally proposed in 1995 by Gonzalez et al. [51], and after being deployed in several current processors such as the popular Intel XScale [19], compiler techniques to exploit it are still in their nascent stages. A horizontally partitioned cache architecture maintains multiple caches at the same level of hierarchy, however each memory address is mapped to exactly one cache. For example, the Intel XScale contains two data caches, a 32KB main cache and a 2KB mini-cache. Each virtual page can be mapped to either of the data caches, depending on the attributes in the page table entry in the data memory management unit. Henceforth we will call the additional cache as the mini-cache and the original cache as the main cache.

The original idea behind such cache organization is the observation that array accesses in loops often have low temporal locality. Each value of an array is used for a while and then not used for a long time. Such array accesses sweep the cache and evict the existing data (like frequently accessed stack data) out of the cache. The problem is worse for high associativity caches that typically employ FIFO page replacement policy. Mapping such array accesses to the small mini-cache reduces the pollution in the main cache and prevents thrashing, thus leading to performance improvements. Thus a horizontally partitioned cache is a simple, yet powerful architectural feature to improve performance. Consequently most existing approaches for partitioning data between the horizontally partitioned caches aim at improving performance.

In addition to performance improvement, horizontally partitioned caches also result in a reduction in the energy consumption due to two effects. First, reduction in the total number of misses results in reduced energy consumption. Second, since the size of the mini-cache is typically small, the energy consumed per access in the mini-cache is less than that in the large main cache. Therefore diverting some memory accesses to the mini-cache leads to a decrease in the total energy consumption. Note that the first effect is in-line with the performance goal and was therefore targeted by traditional performance improvement optimizations. However, the second effect is orthogonal to performance improvement. Therefore energy reduction by the second effect was not considered by traditional performance oriented techniques. In fact, as we show later, the second effect (of a smaller mini-cache) can lead to energy improvements even in the presence of slight performance degradation. Note that this is where the goals of performance improvement and energy improvement diverge.

5.2 Motivation

We illustrate the difference between optimization for performance and energy for horizontally partitioned caches using the memory architecture shown in Figure 5.2(a). The



(a)

```
char a[1024]        // page A
char b[1024]        // page B

for (i=0; i<1024; i++)
    c += a[i] + b[i%5];
```

(b)

Main Cache	Mini Cache	Main Cache Accesses	Main Cache Misses	Mini Cache Accesses	Mini Cache Misses	Runtime (cycles)	Energy (nJ)
A, B	-	2048	65	0	0	3348	8094
A	B	1024	64	1024	1	3348	6046
B	A	1024	1	1024	64	3348	6046
-	A, B	0	0	2048	96	3968	4928

(c)

Figure 5.2: Motivating Example

example architecture has a horizontally partitioned, direct mapped 8KB and 1KB caches, with line size of 16 bytes and other parameters as shown in Figure 5.2(a). We assume that the page size is 1KB, and that each page can be mapped to either of the caches.

Now we examine the execution of the code in Figure 5.2(b) on this architecture. The loop accesses two arrays *a* and *b*. Assuming that the arrays are aligned to the beginning of the page, they occupy two pages (A and B, one page each). To simplify the analysis, we consider only accesses to these two arrays and evaluate the memory latency (total time spent in memory accesses) and the memory energy consumption.

The table in Figure 5.2(c) shows the memory latency (in number of cycles) and energy consumption of the memory subsystem for each of the four possible partitions of the pages. When both the pages are mapped to the main 8KB cache, all the 2048 accesses will be to the main cache, and there will be $64 + 1 = 65$ cold misses. We estimate the performance of the memory subsystem as $2048 + 65 \times 20 = 3348$ cycles, and the energy consumed in the memory subsystem as $2048 \times 3 + 65 \times 30 = 8094$ nJ. Similarly we compute the memory latency and memory energy consumption of the other page partitions. The interesting partition is the last one, in which both the arrays are mapped to the small 1KB mini-cache. For this partition, there are more misses in the mini-cache, but less misses in the main cache. The increase in the misses in the mini-cache is more than the decrease in the misses in the main cache. Therefore there is a performance degradation. However the increase in the energy consumption due to the increased misses in mini-cache is less than the decrease in the energy consumption due to the reduced misses in the main cache, resulting in reduced energy consumption.

Owing to the difference in energy per access between the caches, and the high hit rates of the caches, it intuitively seems that the partition that results in minimum memory energy consumption (optimal energy partition) will have many more pages pages mapped to the mini-cache than the partition that has the least memory latency (optimal performance partition). Thus optimizing for performance is not the same as optimizing for energy. This together with the fact that energy is becoming an ever important design criterion motivates the need for techniques to find optimal energy partitions. In this chapter we propose and evaluate several data partitioning algorithms that aim at minimizing memory subsystem energy consumption in horizontally partitioned cache architectures. It turns out that very simple greedy heuristics work well to optimize energy, and furthermore optimal energy partitions do not suffer significant performance degradation.

5.3 Related Work

Caches are one the major contributors of not only system power and performance, but also of the embedded processor area and cost. In the Intel XScale caches comprise approximately 90% of the transistor count, 60% of the area, and consume approximately 15% of the processor power [47]. As a result several hardware, software and cooperative techniques have been proposed to improve the effectiveness of caches.

Originally horizontally partitioned caches were proposed by Gonzalez et al. [51] to separate and thus reduce the interference between the array and stack variables. Most previous research focuses on achieving performance improvements using horizontally partitioned caches. Hardware based approaches improve the performance by partitioning the data based on reuse history. While some approaches use effective address reuse information [52, 53], others use program counter reuse information [54]. Software based approaches attempt to improve performance primarily by coarse-grain region based scheduling [55, 56]. Such schemes simply map the stack data, or the scalar variables to the mini-cache. Recently Xu et al. [57] proposed a profile-based technique to partition the virtual pages to different caches. Their algorithm has complexity O(m^3n), where m is the number of pages accessed in the application, and n is the number of memory accesses. It should be noted that O(n) is also the time complexity of simulation of an application with a given page mapping.

However, the main focus of all the previous research has been performance improvement, and they achieve energy reduction only as a by-product. In this chapter we show that optimizing for performance does not effectively optimize energy. With energy consumption becoming an ever important design constraint, there is a need to develop data partitioning techniques aimed at energy improvement. We also propose and evaluate several such data partitioning schemes, and demonstrate that in contrast to high-complexity data partitioning techniques for performance improvements proposed earlier, low-complexity techniques work well for energy optimization with negligible performance penalty.

5.4 Our HPC Compiler

The problem of energy optimization for horizontally partitioned caches can be translated into a data partitioning problem. The data memory that the program accesses is divided into pages, and each page can be independently and exclusively mapped to exactly one of the caches. The compiler's job is then to find the mapping of the data memory pages to the caches that leads to minimum energy consumption.

As shown in Figure 5.3, we first compile the application and generate the executable. The *Page Access Information Extractor* calculates the number of times each page is accessed during the execution of the program. Then it sorts the pages in decreasing order of accesses to the pages. The complexity of simulation used to compute the number of accesses to each page and sorting the pages is O($n + m \log(m)$), where n is the number of data memory accesses, and m is the number of pages accessed by the application.

The *Data Partitioning Heuristic* finds the best mapping of pages to the caches that minimizes the energy consumption of the target embedded platform. The *Data Partitioning*

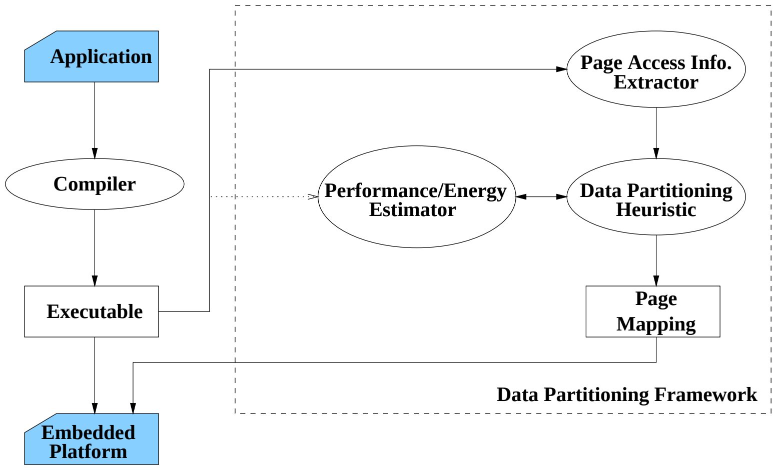

Application
Compiler
Executable
Embedded Platform
Page Access Info. Extractor
Performance/Energy Estimator
Data Partitioning Heuristic
Page Mapping
Data Partitioning Framework

Figure 5.3: Compiler Framework for HPC Data Partitioning

Heuristic can be tuned to obtain the best performing, or minimal energy data partition by changing the cost function *Performance/Energy Estimator*.

The executable together with the page mapping are then loaded by the operating system of the target platform for optimized execution of the application.

5.4.1 Data Partitioning Algorithms

The compiler framework depicted in Figure 5.3 supports using, evaluating, and comparing several data partitioning heuristics of varying complexities to exploit horizontally partitioned caches. All the data partitioning heuristics take as input a list of memory pages accessed by the application, sorted in the order of decreasing accesses to the pages.

```
Heuristic OMN(Pages P)
01: $M = \phi, m = \phi, U = P$
02:  while ($U \neq \phi$)
03:    $p = U.pop()$
04:    $cost1 = evaluatePartitionCost(M + p, m)$
05:    $cost2 = evaluatePartitionCost(M, m + p)$
06:     if ($cost1 \leq cost2$) $M+ = p$ else $m+ = p$
07: endwhile
08: return M, m
```

Figure 5.4: Heuristic OMN

The heuristic shown in Figure 5.4 is a greedy approach for solving the data partitioning problem. Initially, M (list of pages mapped to main cache) and m (list of pages mapped to mini-cache) are empty. All the pages are initially undecided, and are in U (line 01). U is a list containing pages sorted in the decreasing order of accesses. The heuristic picks the first page in U, and evaluates the cost of the partition when the page is mapped to the main cache (line 04) and when it is mapped to the mini-cache (line 05). The heuristic finally maps a page to the partition that results in minimum cost (line 06). Depending on whether the function *evaluatePartitionCost(M, m)* estimates the performance of the partition, or the energy consumption, the heuristic can be used to find the best performing partition, or the minimum energy partition.

evaluatePartitionCost(M, m) uses simulation to estimate the performance or the energy consumption for a given partition. Since each page is considered only once, the complexity of this heuristic is O(mn), where O(n) is the complexity of the simulation-based estimation.

```
Heuristic OM2N(Pages P)
01: $M = \phi, m = \phi, U = P$
02: while ($U \neq \phi$)
03:   $p = U.pop()$
04:   $M1 = M + p, m1 = m, U1 = U$
05:   while ($U1 \neq \phi$)
06:     $p' = U1.pop()$
07:     $cost1' = evaluatePartitionCost(M1 + p', m1)$
08:     $cost2' = evaluatePartitionCost(M1, m1 + p')$
09:     if ($cost1' \leq cost2'$) $M1 += p'$ else $m1 += p'$
10:   endwhile
11:   $M2 = M, m2 = m + p, U2 = U$
12:   while ($U2 \neq \phi$)
13:     $p' = U2.pop()$
14:     $cost1' = evaluatePartitionCost(M2 + p', m2)$
15:     $cost2' = evaluatePartitionCost(M2, m2 + p')$
16:     if ($cost1' \leq cost2'$) $M2 += p'$ else $m2 += p'$
17:   endwhile
18:   $cost1 = evaluatePartitionCost(M1, m1)$
19:   $cost2 = evaluatePartitionCost(M2, m2)$
20:   if ($cost1 \leq cost2$) $M += p$ else $m += p$
21: endwhile
22: return M, m
```

Figure 5.5: Heuristic OM2N

Figure 5.5 describes a higher complexity heuristic, OM2N. For each page $p \in U$ (line 06), this heuristic uses the heuristic OMN to find whether the page should be mapped to the main cache (lines 04-10) or to the mini-cache (lines 11-17). Therefore the complexity of the heuristic OM2N is O(m^2n).

```
Heuristic ON(Pages P)
01: $M = \phi, m = \phi$
02: for ($i = 0; i < \frac{mini-cache\_size}{page\_size}; i++$)
03:     $m += U.pop()$
05: endFor
06: $M = U$
07: return M, m
```

Figure 5.6: Heuristic ON

Our experimental results later show that OMN heuristic works very well for energy reduction. Motivated by this, we developed an even simpler heuristic for data partitioning. Figure 5.6 shows a very simple single step heuristic. If we define $k = \frac{mini-cache\_size}{page\_size}$, then the first k pages with the maximum number of accesses are mapped to the mini-cache, and the rest are mapped to the main cache. This partition aims to achieve energy reduction while making sure that there is no performance loss (for high associativity mini-caches). Note that for this heuristic we do not need to sort the whole list of pages, thus the time complexity of this heuristic is O$(n + km)$, which is O(n), since both k and m are very small as compared to n.

5.5 Effectiveness of Energy-oriented Compilation for HPCs

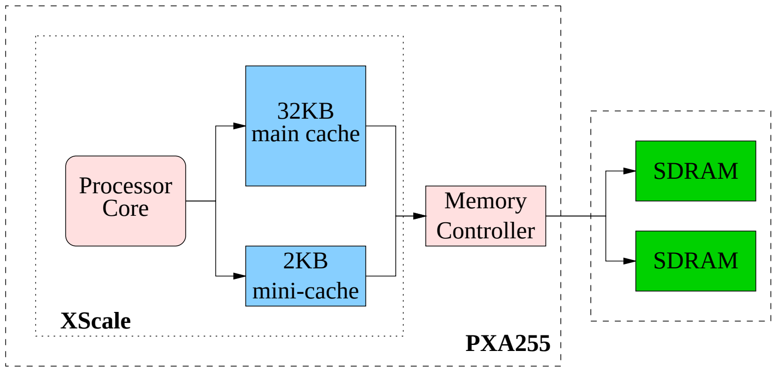


Figure 5.7: Modeled memory subsystem

We have developed a framework that employs data partitioning algorithms described in Section **??** to optimize the memory latency or the memory subsystem energy consumption of applications. We have modified the *sim-safe* simulator from SimpleScalar toolset [58] to obtain the number of accesses to each data memory page. This implements our *Page*

Access Information Extractor in Figure 5.3. In order to estimate the performance/energy of an application for a given mapping of data memory pages to the main cache and the mini-cache, we have developed performance and energy models of memory subsystem of a popular PDA, HP iPAQ h4300 [59].

Figure 5.7 shows the memory subsystem of the iPAQ that we have modeled. The iPAQ uses the Intel PXA255 processor [60] with the XScale core [19], which has a 32KB main cache and 2KB mini-cache. PXA255 also has an on-chip memory controller that communicates with PC100 compatible SDRAMs via off-chip bus. We have modeled the low-power 32MB Micron MT48V8M32LF [61] SDRAM as the off-chip memory. Since the iPAQ has 64MB of memory, we have modeled two SDRAMs.

We use the memory latency as the performance metric. We estimate the memory latency as $(A_m + A_M) + MP \times (M_m + M_M)$, where A_m and A_M are the number of accesses, and M_m and M_M are the number of misses in the mini-cache and the main cache respectively. We obtain these numbers using the *sim-cache* simulator [58], modified to model horizontally partitioned caches. The miss penalty MP was estimated as 25 processor cycles, taking into account the processor frequency (400 MHz), memory bus frequency (100 MHz), the SDRAM access latency in power-down mode (6 memory cycles), and the memory controller delay (1 processor cycle).

We use the memory subsystem energy consumption as the energy metric. There are three components in our estimate of memory energy consumption: energy consumed by the caches, energy consumed by off-chip busses, and the energy consumed by the main memory (SDRAMs). We compute the energy consumed in the caches using the access and miss statistics from the modified sim-cache results. The energy consumed per access for each of the caches is computed using eCACTI [62]. As compared to CACTI [63], eCACTI provides better energy estimates for high associativity caches, since it models sense-amps more accurately and scales device widths according to the capacitive loads. We have used linear extrapolation on cache size to estimate energy consumption of the mini-cache, since both CACTI and eCACTI do not model caches with less than 8 sets.

We use the PCB and layout recommendations of PXA255 and Intel 440MX chipset [64] and the relation between Z_o, C_o and ϵ_r [65], to compute the the energy consumed by the external memory bus in a read/write burst as shown in Table 5.1.

We used the parameters shown in Table 5.2 from the MICRON MT48V8M32LF

Input pin capacitance	3.5 pF
Input/output pin capacitance	5 pF
Bus wire length	2.6 in
PCB characterisitc Impedance, Z_o	60 Ω
Relative permitivity of PCB dielectric, ϵ_r	4.4
Capacitance per unit length, C_o	2.34 pF/in
Capacitance per trace	6.17 pF
Bus energy per burst	9.46 nJ

Table 5.1: External Memory Bus Parameters

SDRAM current I_{dd}	100 mA
SDRAM supply voltage V_{dd}	2.5 V
Memory bus frequency f_{mem}	100 MHz
Number of memory cycles/burst N_{cyc}	13
SDRAM energy per read/write burst E_{mbst}	32.5 nJ

Table 5.2: SDRAM Energy Parameters

SDRAM to compute the energy consumed by the SDRAM per read/write burst operation (cache line read/write), also shown in Table 5.2.

We perform our experiments on applications from MiBench suite [20] and an implementation of H.263 encoder [66]. To compile our benchmarks we used GCC with all optimizations turned on.

5.5.1 Optimizing for Energy is DIFFERENT than Optimizing for Performance

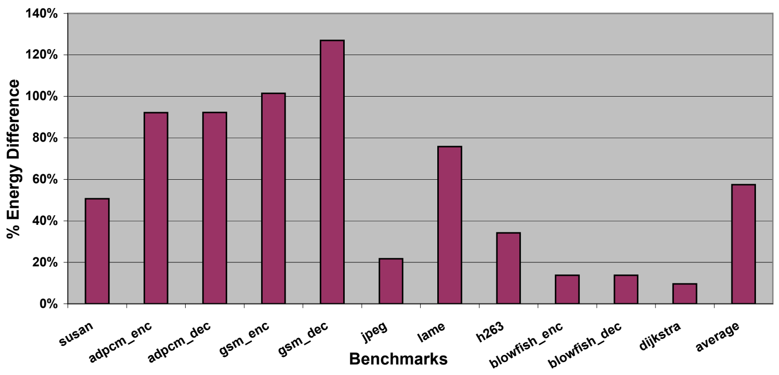


Figure 5.8: Difference in energy consumption between the best energy partition and the best performing partition

Our first experiment investigates the difference in optimizing for energy and optimizing for performance on the memory subsystem. We find the partition that results in the least memory latency, and the partition that results in the least energy consumption. Figure 5.8 plots $\frac{E_{br}-E_{be}}{E_{be}}$, where E_{br} is the memory subsystem energy consumption of the partition that results in least memory latency, and E_{be} is the memory subsystem energy consumption by the partition that results in least memory subsystem energy consumption. For the first five benchmarks (*susan* - *gsm_dec*), the number of pages in the footprint were small, so we could explore all the partitions. For the last seven benchmarks (*jpeg* - *dijkstra*), we took the partition found by the OM2N heuristic as the best partition. As we present later, OM2N gives close-to-optimal results in the cases when we were able to search optimally. The graph essentially plots the increase in energy if you choose the best performance partition as your design point. The increase in the energy consumption is up to 130% and on average 58% for this set of benchmarks.

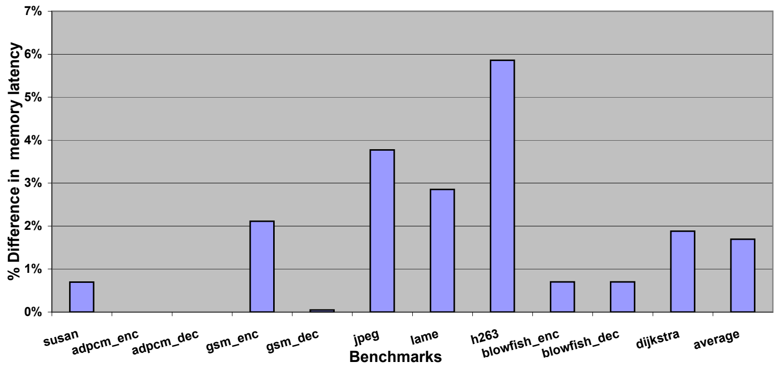


Figure 5.9: Difference in runtime between the best performance partition and the best energy partition

Figure 5.9 plots $\frac{R_{be}-R_{br}}{R_{be}}$, where R_{be} is the memory latency (in cycles) of the best energy partition, and R_{br} is the memory latency of the best performing partition. This graph shows the increase in memory latency when you choose the best energy partition, as compared to using the best performance partition. The increase in memory latency is on average 1.7% and 5.8% in the worst case for this set of benchmarks. Thus choosing the best energy partition results in significant energy savings at a minimal loss in performance.

5.5.2 Simple Greedy Heuristics work well for Energy Optimization

Next we evaluate the effectiveness of various data partitioning heuristics. Figure 5.10 plots $\frac{E_{base}-E_{min}}{E_{base}}$ for each heuristic, where E_{base} is the energy consumed in the base case,

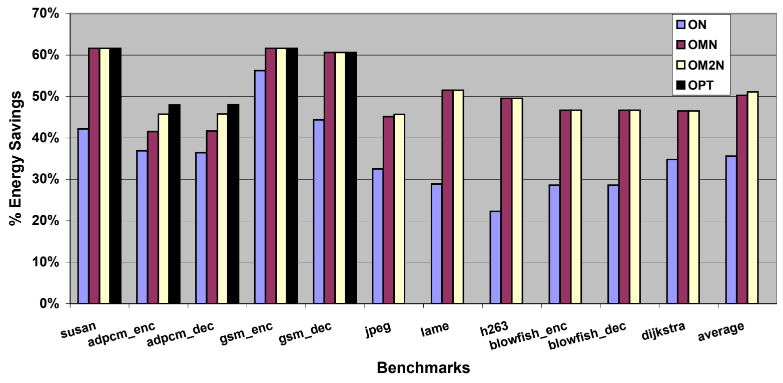


Figure 5.10: Energy savings achieved by various heuristics

i.e. when all the data is mapped to the main cache. No page is mapped to the mini-cache, and E_{min} is the energy of the minimum energy partition as computed by the heuristic. The rightmost black bars in the first five benchmarks (*susan* - *gsm_dec*) represent the energy savings achieved by the optimal search. The optimal search can achieve on average 55% savings in the energy consumed by the memory subsystem. It can also be seen that for these benchmarks, heuristic OM2N achieves close-to-optimal (within 2%) results. Thus we did not feel the need to investigate more complex heuristics. Instead we developed low-complexity heuristics aimed at achieving energy savings. The OMN heuristic achieves up to 62% and on average 50% savings in the energy consumed by the memory subsystem. Thus simple greedy heuristics work well to reduce the energy consumption. These heuristics have lower complexity than the O(m^3n) heuristic suggested in [57] for performance improvement.

In practice, such high complexity algorithms are unlikely to be used due to their large runtimes. For example, for the *jpeg* benchmark, the optimal algorithm would take more than 5,000 years, our OM2N heuristic takes a few days, the OMN heuristic takes a few hours, and the ON heuristic takes a few minutes to find the optimal partition. Clearly there is a need for low complexity, yet effective data partitioning heuristics.

Furthermore as we have noted that the energy optimal partitions incur minimal loss in performance, therefore for several design domains, it makes sense to use the energy optimal partition. The energy optimal partition saves significant energy at minimal cost. Even the single step ON heuristic is also able to achieve up to 57% and on average 35% energy reduction.

Figure 5.11 plots the goodness of the ON and OMN heuristic in obtaining energy reduction. The goodness of a heuristic is defined as energy reduction achieved by the heuristic

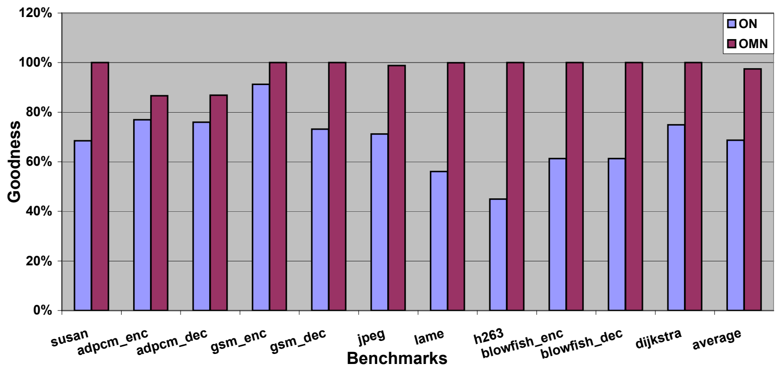


Figure 5.11: Goodness of various heuristics

as compared to the maximum energy reduction that was possible, i.e., $\frac{(E_{Main}-E_{alg})}{(E_{Main}-E_{best})}$, where E_{Main} is the energy consumption when all the pages are mapped to the main cache, E_{alg} is the energy consumption of the best energy partition that the heuristic found and E_{best} is the energy consumption of the best energy partition. For the last seven benchmarks for which we could not perform the optimal search, we assume the partition found by the heuristic OM2N as the best energy partition. The graph shows that the OMN heuristic could obtain on average 97% of the possible energy reduction, while ON could achieve on average 64% of the possible energy reduction. It is important to note here that the GCC compiler for XScale does not exploit the mini-cache at all. The ON heuristic provides a simple yet effective way to exploit the mini-cache without incurring any performance penalty (for high associativity mini-caches).

5.5.3 Sensitivity Analysis

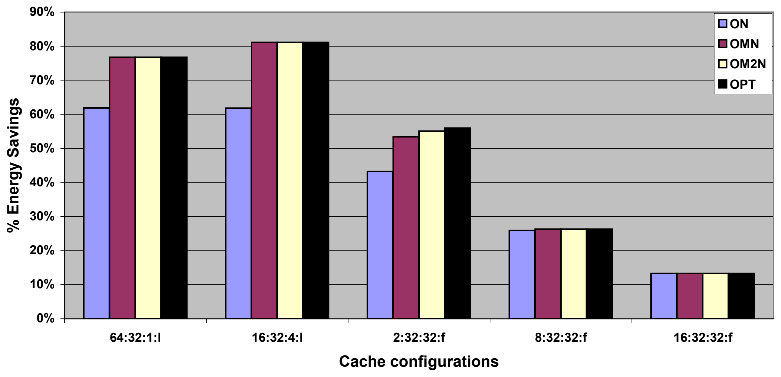


Figure 5.12: Sensitivity analysis

Next we performed sensitivity analysis of our partitioning algorithms to various cache configurations. Figure 5.12 plots the average energy savings achieved by the heuristics across several cache configurations on the first five benchmarks. A cache configuration is specified as *no_of_sets : linesize_in_bytes : associativity : replacement_policy*. The first configuration is a direct mapped 2KB cache, while the second configuration is a 4-way set associative 2KB cache. The third configuration is the original configuration, i.e., a 32-way set associative 2KB cache. The fourth configuration is a 32-way, 8KB cache and the fifth configuration corresponds to a 32-way 16KB cache.

In the first two configurations, we vary the associativity of the cache while keeping the size constant, and in the last two configurations, we vary the size of the cache while keeping the associativity constant. For all cache configurations, the difference between the energy consumption of the minimum energy partition found by OM2N, OMN and the optimal algorithm is no more than 2%.

Decreasing the associativity increases the energy reduction, because of the decrease in the energy per access. But for the direct mapped cache, the performance begins to deteriorate, thereby increasing the energy consumption. As we increase the cache size, the energy per access of the mini-cache increases, therefore there is no improvement in the total achievable energy reduction. The best mini-cache configuration for the energy reduction in horizontally partitioned caches for our set of benchmark is the second configuration, i.e., a 4-way set associative 2KB cache.

Thus although the energy gains achieved by changing cache parameters vary, greedy heuristics are able to consistently achieve near-optimal results.

5.6 Compiler-in-the-Loop Exploration of HPCs

HPC is a simple, yet very effective architectural feature to improve performance and power consumption of processor systems. However, as we discover in this chapter, the energy reduction obtained using HPCs is very sensitive on the HPC design parameters. Therefore there is much scope for power and performance improvement by customizing the HPC cache configuration to the application set intended for the system.

Since traditionally caches have been significant contributors to the power and performance of the system, there has been extensive previous research to improve the power and

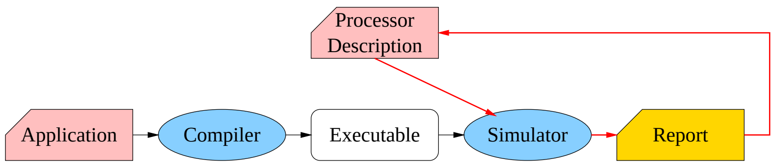


Figure 5.13: Simulation-only Processor Design Exploration

performance of processor systems by tuning the cache parameters (cache size, associativity etc.) to the intended application set. However, since the compiler can be oblivious of the presence of caches, existing cache exploration techniques do not include compiler-in-the-loop of exploring the cache configurations. Traditional exploration is a SO DSE, i.e., compiler is not in the exploration loop is depicted in Figure 5.13. The application is compiled once and an executable is generated. The executable is then simulated over various cache configurations to find out which is the best one.

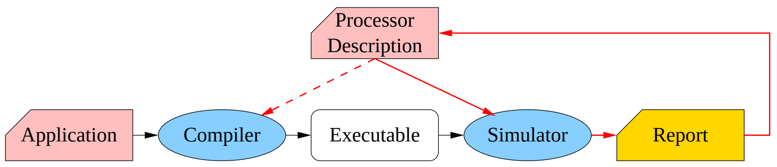


Figure 5.14: Simulation-only Processor Design Exploration

However, to effectively utilize the HPC architecture, the compiler needs to be aware of the HPC cache configuration. Acknowledging the impact of HPC-aware compilers in reducing the energy consumption of the memory susbsystem, we propose a Compiler-in-the-Loop (CIL) exploration of HPC configurations, as depicted in Figure 5.14.

To accurately evaluate an HPC configuration, the application has to be first compiled for that HPC configuration. The executable generated is then simulated on the HPC configuration to accurately estimate the power, and performance corresponding to the HPC configuration.

5.6.1 HPC Exploration Framework

Figure 5.15 is the high-level view of our Compiler-in-the-Loop exploration methodology for HPC design. The framework is driven by a processor architecture description. We use EXPRESSION [3] ADL to describe the processor and drive our exploration. It should be noted however, that the technique we present is generic and is not restricted on any

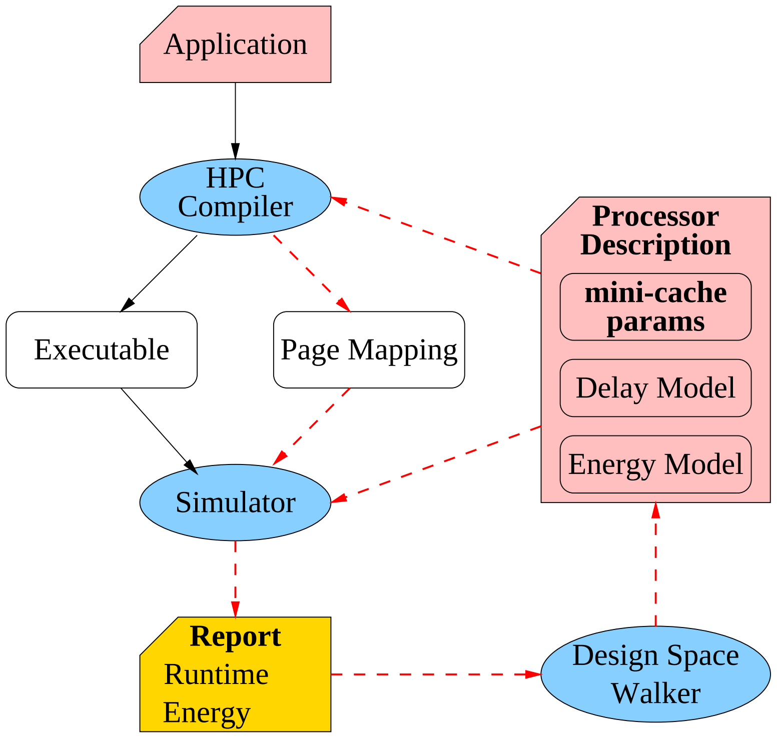


Figure 5.15: Compiler-in-the-Loop methodology to explore the design space of HPCs

particular ADL. We need three kinds of information from the processor description: i) the Horizontally Partitioned Cache parameters, ii) the energy models, and iii) the delay model of the memory subsystem of the processor. The HPC compiler is parameterized on these architectural features. We employ the energy-oriented OMN compilation technique [67] to generate the page mapping which will result in minimum energy consumption by the memory subsystem for the given application and HPC configuration.

5.6.2 Design Space

We specify the mini-cache using two attributes, the mini-cache size and the mini-cache associativity. For our experiments, we vary cache size from 256 bytes to 32 KB, in exponents of 2. The lower bound comes from the restriction of eCACTI to provide energy model for caches of size less than 256 bytes. The upper bound is 32 KB, which is the size of the main cache in XScale. We explore the whole range of mini-cache associativities, i.e., from direct mapped to fully associative. We do not model mini-cache configurations which cannot be modeled by eCACTI (as well as by CACTI). We set the line size to be as in the Intel XScale architecture (32 bytes). In total we explore 33 mini-cache configurations for each benchmark.

5.6.3 Importance of Exploration

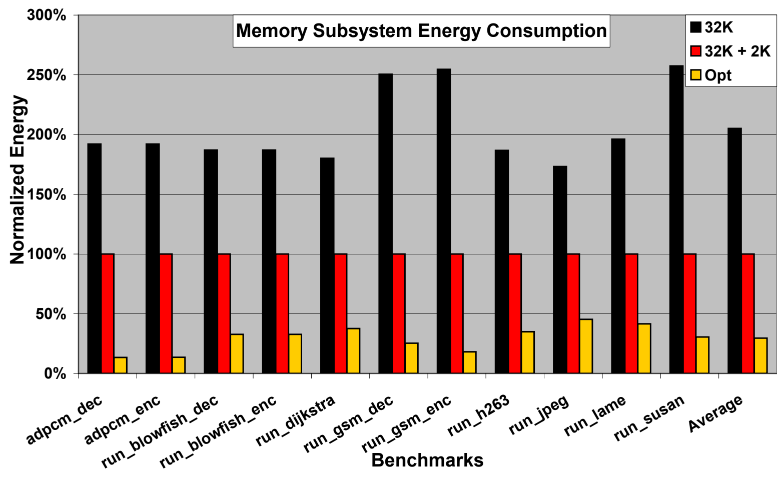


Figure 5.16: Energy savings achieved by exploration

We first present experimental results to demonstrate the importance of exploring HPCs. Figure 5.16 plots three bars for applications for each benchmark. The last set of bars is the average over the applications. The leftmost bar represents the energy consumed by the memory subsystem when the system has only a 32 KB main cache (no mini-cache is present.) The second, or the middle bar shows the energy consumed when there is a 2 KB mini-cache in parallel with the 32 KB cache, and the application is compiled to achieve minimum energy. The third and the rightmost bar represents the energy consumed by the memory subsystem, when the mini-cache parameters (size and associativity) are chosen using exhaustive Compiler-in-the-Loop exploration. All the energy values are normalized to the case when there is a 2 KB mini-cache (the Intel XScale configuration.)

We make two important observations from this graph. The first is that Horizontally Partitioned Caches are very effective in reducing the memory subsystem energy consumption. As compared to using not using any mini-cache, using default mini-cache (the default mini-cache is 2 KB, 32-way set associative) leads to 2X reduction in energy consumption on average. The second important observation is that the energy reduction obtained is very sensitive on the mini-cache parameters. Compiler-in-the-Loop exploration of the mini-cache design space to find the minimum energy mini-cache results in additional 80% energy reduction on average, thus reducing the energy consumption to 20% from the case with a 2

KB mini-cache.

Furthermore, the performance of the energy-optimal HPC configuration is very close to the performance of the best performing HPC configuration. The performance degradation was no more than 5% and was 2% on average. Therefore energy-optimal HPC configuration achieves high energy reductions at negligible performance cost.

Figure 5.3 shows the energy optimal mini-cache configuration for each benchmark. The table suggests that low-associativity mini-caches are good candidates to achieve low-energy solutions.

Benchmark	**mini-cache Parameters**
adpcm_dec	8K, direct mapped
adpcm_enc	4K, 2-way
dijkstra	8K, 2-way
blowfish_dec	16K, 2-way
blowfish_enc	16K, 2-way
gsm_dec	2K, direct mapped
gsm_enc	2K, direct mapped
h263	8K, 2-way
jpeg	16K, 2-way
lame	8K, 2-way
susan	2K, 4-way

Table 5.3: Optimal mini-cache parameters

5.6.4 Importance of CIL DSE

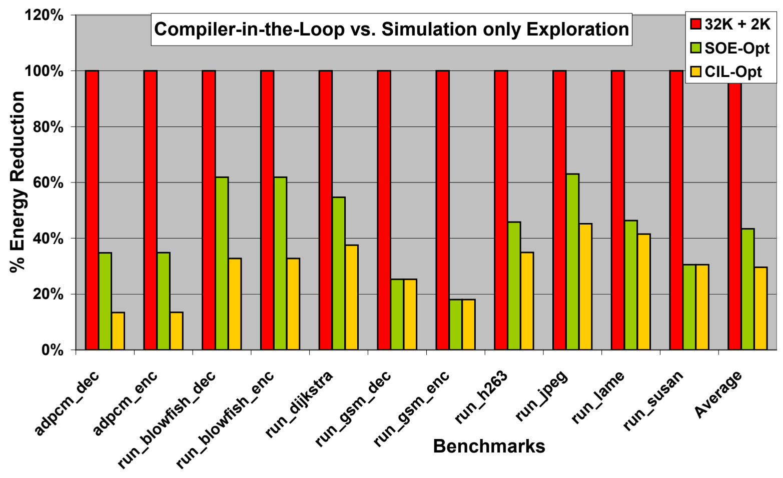


Figure 5.17: CIL versus SO exploration

```
OptimalExploration()
01: $minEnergy = MAX\_ENERGY$
02: foreach $(c \in C)$
03:   $energy = estimateEnergy(c)$
04:   if $(energy < minEnergy)$
05:     $minEnergy = energy$
06:   endIf
07: endFor
08: return minEnergy
```

Figure 5.18: Optimal Exploration Algorithm

In this section we show the importance of performing CIL DSE and show its superiority to the traditional SO DSE approach. To this goal, we perform cache configuration exploration using both approaches. Figure 5.17 shows the best achieved energy reductions for each benchmark using CIL DSE (right bar) and SO DSE approach (middle bar). In the latter case the page mapping was determined once for each benchmark (the energy-optimal mapping for 32K/2K horizontally partitioned cache) and was used for all other cache configurations. All values are normalized to the 32K/2K cache energy consumption. Figure 5.17 demonstrates that CIL DSE is able to find configurations that result in average 33% lower energy consumption than SO DSE.

5.6.5 Exploration Algorithms

We have demonstrated that exploring the mini-cache design space is very important, and significant energy savings can be obtained by correctly choosing the cache parameters. However since the mini-cache design space is very large, exhaustive exploration may consume a lot of time. In this section we present some techniques to explore the mini-cache design space to reduce the exploration-time.

5.6.5.1 Exhaustive Algorithm

Figure 5.18 describes the optimal exploration algorithm. The algorithm estimates the energy consumption for each mini-cache configuration (line 02), and keeps track of the minimum energy. The function *estimate_energy*, estimates the energy consumption for a given mini-cache size and associativity.

We perform exhaustive exploration to figure out the mini-cache parameters which results in minimum energy consumption. The rightmost bar in Figure 5.19 plots the energy consumption of the optimal configuration, as compared to the energy consumption when the XScale default 32-way, 2K mini-cache is used. The rightmost bar in Figure 5.20 plots the time (in hours) required to explore the design space using the optimal algorithm.

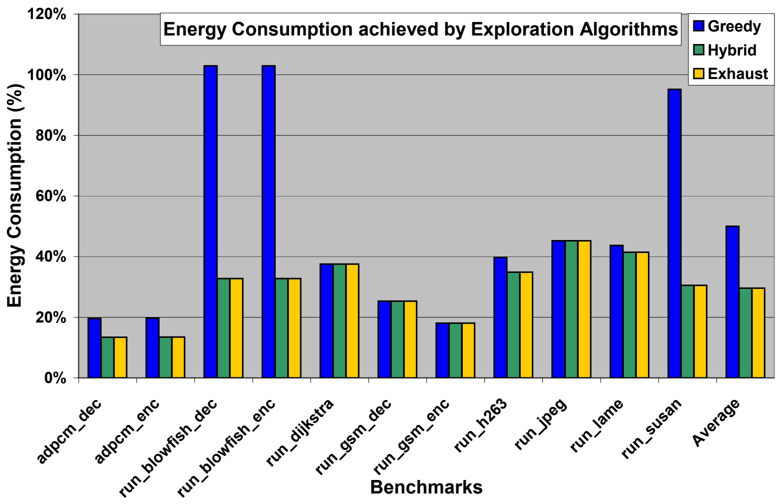


Figure 5.19: Relative energy consumption achieved by exploration algorithms

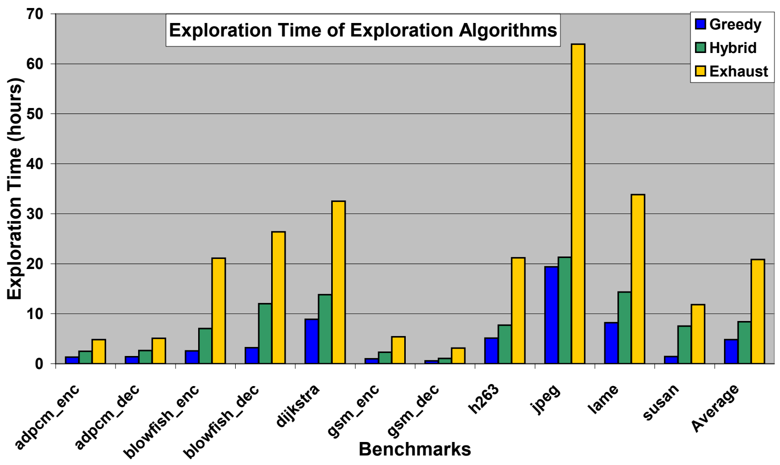


Figure 5.20: Exploration time of exploration algorithms

5.6.5.2 Greedy Algorithm

In an attempt to reduce the runtime of the exhaustive algorithm, we developed a very simple **greedy exploration algorithm** to explore the mini-cache design space. The greedy algorithm outlined in Figure 5.10 first greedily finds the cache size (lines 02-04), and

```
GreedyExploration()
01: $size = MIN\_SIZE, assoc = MIN\_ASSOC$

// greedily find the size
02: while ($betterNewConf(size \times 2, assoc, size, assoc)$)
03:     $size = size \times 2$
04: endWhile

// greedily find the assoc
05: while ($betterNewConf(size, assoc \times 2, size, assoc)$)
06:     $assoc = assoc \times 2$
07: endWhile

08: return $estimateEnergy(size, assoc)$


betterNewConf(size', assoc', size, assoc)
01: if ($!existsCacheConfig(size', assoc')$)
02:     return $false$
03: $energy = estimateEnergy(size, assoc)$
04: $energy' = estimateEnergy(size', assoc')$
05: return ($energy' < energy$)
```

Figure 5.21: Greedy Exploration Algorithm

then greedily finds the associativity (lines 05-07). The function *betterNewConfiguration* tells whether the new mini-cache parameters result in lower energy consumption than the old mini-cache parameters.

The middle bar in Figure 5.19 plots the energy consumption when the mini-cache configuration is chosen by the greedy algorithm, and the leftmost bar in Figure 5.20 plots the time that greedy exploration requires to explore the design space of the mini-cache. Although the greedy algorithm reduces the exploration time on an average by a factor of 5X, the energy consumption is on an average 2X more than what is achieved by an exhaustive algorithm. A closer look into the graph reveals that even the greedy algorithm finds out the near-optimal mini-cache configuration in most of the cases. Clearly, there is a trade-off between the exploration time and the energy reductions that can be obtained thereby.

```
HybridExploration()
01: $size = MIN\_SIZE, assoc = MIN\_ASSOC$

// greedily find the size
02: while ($betterNewConf(size \times 4, assoc, size, assoc)$)
03:   $size = size \times 4$
04: endWhile

// search in the neighbourhood
05: $done = false$
06: while ($!done$)
07:   if ($betterNewConf(size \times 2, assoc, size, assoc)$)
08:     $size = size \times 2$
09:   else if ($betterNewConf(size, assoc \times 2, size, assoc)$)
10:     $assoc = assoc \times 2$
11:   else if ($betterNewConf(size \div 2, assoc, size, assoc)$)
12:     $size = size \div 2$
13:   else if ($betterNewConf(size \div 2, assoc \div 2, size, assoc)$)
14:     $size = size \div 2, assoc = assoc \div 2$
15:   else if ($betterNewConf(size \div 2, assoc \times 2, size, assoc)$)
16:     $size = size \div 2, assoc = assoc \times 2$
17:   else
18:     $done = true$
19: endWhile

20: return $estimateEnergy(size, assoc)$
```

Figure 5.22: Hybrid Exploration Algorithm

5.6.5.3 Hybrid Algorithm

To obtain best of both worlds, we developed a hybrid algorithm that can achieve energy consumption very close to the exhaustive configuration, while consuming time closer to the greedy algorithm. Figure 5.22 outlines the hybrid algorithm. The hybrid algorithm first greedily searches for the optimal mini-cache size (lines 02-04). Note however that it tries every alternate mini-cache size. The hybrid algorithm tries mini-caches sizes in exponents of 4, rather than 2 (line 03). Once it has found the optimal mini-cache size, then it explore exhaustively in the size-associativity neighborhood (lines 07-15) to find a better size-associativity configuration.

The leftmost bar in Figure 5.19 plots the energy consumption when the mini-cache configuration is chosen by the hybrid algorithm, and the leftmost bar in Figure 5.20 plots the time that hybrid exploration requires to explore the design space of the mini-cache. Our hybrid algorithm is able to find the optimal mini-cache configuration in all of our benchmarks, while it takes about 3X less time than the exhaustive algorithm.

5.7 Summary

Memory may be the single largest consumer of processor energy. As a result several existing research efforts have focused on developing techniques to reduce the energy consumption of the memory.

In this chapter we focused on developing a novel compilation technique to reduce memory energy consumption by exploiting the Horizontally Partitioned Cache architectural (HPC) feature, popular in embedded processors, e.g., StrongARM 11000, and the Intel XScale. HPC is a memory architectural feature in which processors have multiple caches (typically two) at the same level of memory hierarchy. If the application data is wisely partitioned into the two caches, cache pollution can be reduced and therefore performance improvements can be achieved. This was the original motivation behind implementing HPCs in processors. Existing data partitioning techniques aim at performance improvement and obtain energy improvements just as a by-product.

Typically one of the caches in the HPC is smaller, and has less energy consumption per access. Mapping data to the smaller cache therefore results in reduction in the energy consumption. However mapping too much data results in increased misses in the smaller cache, and therefore ultimately leads to performance and energy degradation. Therefore it is very important to wisely partition the data between the two caches.

Our data partitioning technique, which aims at energy reduction using HPCs is able to reduce the memory subsystem energy consumption by 50% at on average 3% performance loss. Thus, HPC is a very effective technique to reduce the memory energy consumption. However, the energy reductions obtained are very sensitive to the compiler and the HPC configuration. As a result it is very important to include compiler effects when deciding the HPC configuration. We show that SO DSE results in sub-optimal results. CIL DSE can uncover HPC configurations which have up to 30% lesser energy consumption than the one discovered by the SO DSE, underlining the need and usefulness of CIL DSE.

Chapter 6

Summary and Future Directions

Embedded systems are characterized by stringent, application-specific, multi-dimensional design constraints including power consumption, performance, cost weight and time-to-market. To achieve such flexibility in design, programmable embedded systems, which contain a programmable processor are becoming a popular choice.

However, to meet all the design constraints simultaneously, the embedded processor designs are highly customized. Embedded processors are often designed by removing some costly (in terms of area, power etc) architecture features present in high-end processors, implementing other architectural features only partially, and implementing some new light-weight architectural features.

Consequently, code generation for the embedded processors is a very challenging task. Apart from the lack of compiler technology to exploit the new, or partial architectural features, an embedded system compiler has to avoid the penalties due to missing design features. Even if the compiler is able to do all this, it has to perform all these in the restricted set of resources present on the embedded systems. However, if the compiler is able to exploit the architectural features of the embedded processors, it can make a tremendous difference in the power, performance, etc. of the eventual system.

However, existing embedded system design/exploration techniques either do not consider the compiler effects on the design, or include the compiler effects in an ad-hoc manner. This may lead to inaccurate evaluation of design choices and therefore lead to suboptimal design decisions.

This thesis proposed a *Compiler-in-the-Loop* (CIL) Design Space Exploration (DSE) methodology – a systematic method to include compiler effects during architectural evaluation of embedded systems. This dissertation demonstrates the need and usefulness of the proposed methodology at several levels of embedded processor design.

At the processor pipeline design abstraction, we developed a novel compiler technique to exploit the partial bypasses present in embedded systems, and improve performance by up to 20%. We further demonstrate that not including compiler effects in DSE, or SO DSE can result in inaccurate evaluation of processor configurations, leading to sub-optimal design decisions. This establishes the need and usefulness of CIL DSE.

At the instruction set architecture design abstraction, we first develop a novel compilation technique for rISA instruction set architectures that can consistently achieve 35% code compression. Thus using CIL DSE of the rISA design space we demonstrate that careful rISA design using CIL DSE can double the achievable code compression.

At the processor pipeline - memory interface design abstraction, we develop a technique to aggregate the naturally occurring processor stalls (due to cache misses) into a large stall so that the processor can be switched to low-power IDLE state and reduce processor power consumption by up to 18%. Not considering this processor power reduction effects of our technique during SO DSE results in significant overestimation of processor power consumption, which may again lead to incorrect design decision; thus establishing the efficacy of CIL DSE.

Finally, at the memory design abstraction, we first develop data partitioning techniques that can reduce the memory energy consumption by 50% using the popular HPC memory architectural feature. Considering our compiler technique while designing HPC configuration can discover HPC configurations with 33% lower memory energy consumption.

In this dissertation, we have extended the compiler technology by developing new architecture-sensitive compilation techniques at all processor design abstractions. In addition, we have also demonstrated the need and usefulness of incorporating the effects of our compiler during processor design by CIL DSE. However this is just the beginning. There are many more architectural features at each of these abstractions which the compilers can exploit, and by incorporating the effect of the compiler while designing these architectural features, pareto-optimal processor designs can be generated automatically. The ultimate aim of this endeavor is that if architecture-sensitive compilation techniques can be developed for all the "most important" features of processor, and the compiler effects can be incorporated at design time, then true processor architecture - compiler codesign can be achieved, resulting in an automatic pareto-optimal processor design methodology.

Bibliography

[1] Starcore LLC. *SC1000-Family Processor Core Reference.*

[2] A. Halambi, A. Shrivastava, N. Dutt, and A. Nicolau. A customizable compiler framework for embedded systems. In *In Proceedings of SCOPES*, 2001.

[3] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of Design Automation and Test in Europe*, 1999.

[4] The Trimaran Consortium. *The Trimaran Compiler Infrastructure for Instruction Level Parallelism.*

[5] P. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* 1990.

[6] P. Ahuja, D. W. Clark, and A. Rogers. The performance impact of incomplete bypassing in processor pipelines. In *Proc. of Symposium on Microarchitecture MICRO-28*, 1995.

[7] P. Grun, A. Halambi, N. Dutt, and A. Nicolau. RTGEN: An algorithm for automatic generation of reservation tables from architectural descriptions. In *ISSS*, 1999.

[8] S.S. Muchnick. *Advanced Compiler Design and Implementation.* 1998.

[9] E. Bloch. The engineering design of the stretch computer. In *Proc. of Eastern Joint Computer Conference*, pages 48–59, 1959.

[10] R. Cohn, T. Gross, M. Lam, and P. Tseng. Architecture and compiler tradeoffs for a long instruction word microprocessor. In *Proc. of ASPLOS*, 1989.

[11] A. Abnous and N. Bagerzadeh. Pipelining and bypassing in a vliw processor. In *IEEE trans. on Parallel and Distributed Systems*, 1995.

[12] J. Gray, A. Naylor, A. Abnous, and N. Bagherzadheh. Viper: A vliw integer microprocessor. In *IEEE Journal of Solid State Circuits*, pages 1377–1383, 1993.

[13] M. Buss, R. Azavedo, P. Centoducatte, and G. Araujo. Tailoring pipeline bypassing and functional unit mapping for application in clustered vliw architectures. In *Proc. of CASES*, 2001.

[14] K. Fan, N. Clark, M. Chu, K. V. Manjunath R. Ravindran, M. Smelyanskiy, and S. Mahlke. Systematic register bypass customization for application-specific processors. In *Proc. of ASSAP*, 2003.

[15] E. S. Davidson. The design and control of pipelined function generators. *Int. IEEE Conf. on Systems Networks and Computers*, pages 19–21, 1971.

[16] T. Muller. Employing finite automata for resource scheduling. In *Proc. of Symposium on Microarchitecture MICRO-27*, 1993.

[17] T. A. Proebsting and C. W. Fraser. Detecting pipeline structural hazards quickly. In *Proc. of ACM SIGPLAN-SIGACT Symp. on PLDI*, 1994.

[18] P. G. Lowney, S. M. Freudenberger, T̃. J. Karzes, W. D. Lichtenstein, R̃. P. Nix, J. S. O'Donnell, and J̃. C. Ruttenberg. The Multiflow Trace Scheduling compiler. *The Journal of Supercomputing"*, 7(1-2):51–142, 1993.

[19] Intel Corporation, http://www.intel.com/design/intelxscale/273473.htm. *Intel XScale(R) Core: Developer's Manual.*

[20] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE Workshop in workload characterization*, 2001.

[21] Intel Corporation, http://www.intel.com/design/iio/manuals/273411.htm. *Intel 80200 Processor based on Intel XScale Microarchitecture.*

[22] Aviral Shrivastava, Eugene Earlie, Nikil Dutt, and Alex Nicolau. Operation tables for scheduling in the presence of incomplete bypassing. In *CODES+ISSS '04: Proceedings*

of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, pages 194–199, New York, NY, USA, 2004. ACM Press.

[23] http://www.synopsys.com/products/logic/design_compiler.html. *Synopsys Design Compiler*, 2001.

[24] S. Hanono and S. Devadas. Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator. In *Proc. of Design Automation Conference (DAC)*, 1998.

[25] G. Goosens et al. CHESS: Retargetable Code Generation for Embedded DSP Processors. In *Code Generation for Embedded Processors.* Kluwer, 1997.

[26] Advanced RISC Machines Ltd. *ARM7TDMI (Rev 4) Technical Reference Manual.*

[27] LSI LOGIC. *TinyRISC LR4102 Microprocessor Technical Manual.*

[28] ST Microelectronics. *ST100 Technical Manual.*

[29] ARC Cores. *ARCtangent-A5 Microprocessor Technical Manual.*

[30] L Benini, A. Macii, and A. Nannarelli. Cached-code compression for energy minimization in embedded processors. In *In Proceedings of ISLPED*, 2001.

[31] H. Lekatsas, Jorg Henkel, and Wayne Wolf. Code compression for low power embdedded system design. In *In Proceedings of DAC 2000*, 2000.

[32] A. Shrivastava and N. Dutt. energy efficient code generation exploiting reduced bit-width instruction set architecture. In *In Proceedings of ASPDAC*, 2004.

[33] Young-Jun Kwon, Xiarong Ma, and Hyuk Jae Lee. PARE: instruction set architecture for efficient code size reduction. *Electronics Letters 25th Nov'99 Vol. 35 No. 24*, pages 2098–2099, 1999.

[34] Young-Jun Kwon, Danny Parker, and Hyuk Jae Lee. Toe: Instruction set architecture for code size reduction and two operations execution. In *In Proceedings of CASES*, 1999.

[35] A. Krishnaswamy and R. Gupta. Enhancing the performance of 16-bit code using augmenting instructions. In *In Proceedings of LCTES*, 2003.

[36] S. Novack and A. Nicolau. Resource directed loop pipelining : Exposing just enough parallelism. *In The Computer Journal*, 1997.

[37] A. Nicolau and S. Novack. Trailblazing: A hierarchical approach to percolation scheduling. In *In Proceedings of ICPP*, 1993.

[38] P. Briggs, K.D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. In *In Proceedings of PLDI*, 1994.

[39] A. Khare, N. Savoiu, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. V-SAT: A visual specification and analysis for system-on-chip exploration. In *In Proceedings of EUROMICRO*, 1999.

[40] K. Kissell. MIPS16: High-density MIPS for the embedded market. Technical report, Silicon Graphics MIPS Group, 1997.

[41] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys (CSUR)*, 32(2):174–199, 2000.

[42] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. *SIGARCH Comput. Archit. News*, 23(2):392–403, 1995.

[43] L. Benini, A. Bogliolo, and G. D. Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on VLSI Systems*, 8(3):299–316, 2000.

[44] J. M. Rabaey and M. Pedram. Low power design methodologies. In *Kluwer*, 1996.

[45] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the design of the alpha 21264 microprocessor. In *Design Automation Conference*, pages 726–731, 1998.

[46] J. D. McCuplin. Memory bandwidth and machine balance in current high performance computers. *Newsletter of IEEE Computer Architecture Technical Committee*, 1:19–25, 1995.

[47] L. T. Clark, E. J. Hoffman, M. Biyani, Y. Liao, S. Strazdus, M. Morrow, K. E. Velarde, and M. A. Yarch. An embedded 32-b microprocessor core for low-power and high-performance applications. *IEEE Journal of Solid State Circuits*, 36(11):1599–1608, 2001.

[48] G. Contreras, M. Martonosi, J. Peng, R. Ju, and G. Y. Lueh. XTREM: A power simulator for the intel xscale core. In *LCTES*, 2004.

[49] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of ASPLOS 1991*, 1991.

[50] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 6(2):149–206, 2001.

[51] Antonio Gonzalez, Carlos Aliagas, and Mateo Valero. A data cache with multiple caching strategies tuned to different types of locality. In *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 338–347, New York, NY, USA, 1995. ACM Press.

[52] Jude A. Rivers, Edward S. Tam, Gary S. Tyson, Edward S. Davidson, and Matt Farrens. Utilizing reuse information in data cache management. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 449–456, New York, NY, USA, 1998. ACM Press.

[53] Teresa L. Johnson and Wen mei W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *ISCA*, pages 315–326, 1997.

[54] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. A modified approach to data cache management. In *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture*, pages 93–103, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[55] Hsien-Hsin S. Lee and Gary S. Tyson. Region-based caching: an energy-delay efficient memory architecture for embedded processors. In *CASES '00: Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 120–127, New York, NY, USA, 2000. ACM Press.

[56] Osman S. Unsal, Israel Koren, C. Mani Krishna, and Csaba Andras Moritz. The minimax cache: An energy-efficient framework for media processors. In *HPCA '02: Proceedings of the Eighth International Symposium on High-Performance Computer*

Architecture (HPCA'02), page 131, Washington, DC, USA, 2002. IEEE Computer Society.

[57] Rong Xu and Zhiyuan Li. Using cache mapping to improve memory performance of handheld devices. *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on - ISPASS*, pages 106–114, 2004.

[58] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[59] Hewlett Packard, http://www..hp.com. *HP iPAQ h4000 Series - System Specifications.*

[60] Intel Corporation, http://www.intel.com/design/pca/applicationsprocessors/manuals/278693.htm. *Intel PXA255 Processor: Developer's Manual.*

[61] Micron Technology Inc., http://www.micron.com/products/dram/mobilesdram/. *MICRON Mobile SDRAM MT48V8M32LF Datasheet*, 2005.

[62] Mahesh Mamidipaka and Nikil Dutt. eCACTI: An enhanced power estimation model for on-chip caches. In *Technical Report TR-04-28, CECS, UCI*, 2004.

[63] P. Shivakumar and N. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. In *WRL Technical Report 2001/2*, 2001.

[64] Intel Corporation, http://www.intel.com/ design/mobile/desguide/251012.htm. *LV/ULV Mobile Intel Pentium III Processor-M and LV/ULV Mobile Intel Celeron Processor (0.13u) /Intel 440MX Chipset: Platform Design Guide*, 2002.

[65] *IPC-D-317A Design Guidelines for Electronic Packaging Utilizing High-Speed Techniques*, 1995.

[66] K. Lillevold et al. *H.263 test model simulation software.* Telenor R&D, 1995.

[67] Aviral Shrivastava, Ilya Issenin, and Nikil Dutt. Compilation techniques for energy reduction in horizontally partitioned cache architectures. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 90–96, New York, NY, USA, 2005. ACM Press.