Software Instruction Caching

by

Jason Eric Miller

S.B., Computer Science and Engineering
Massachusetts Institute of Technology, 1999

M.Eng., Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 1999

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2007

Author . . . .
Department of Electrical Engineering and Computer Science
May 25, 2007

Certified by . . . .
Anant Agarwal
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Software Instruction Caching

by

Jason Eric Miller

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2007, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

As microprocessor complexities and costs skyrocket, designers are looking for ways to simplify their designs to reduce costs, improve energy efficiency, or squeeze more computational elements on each chip. This is particularly true for the embedded domain where cost and energy consumption are paramount. Software instruction caches have the potential to provide the required performance while using simpler, more efficient hardware. A software cache consists of a simple array memory (such as a scratchpad) and a software system that is capable of automatically managing that memory as a cache.

Software caches have several advantages over traditional hardware caches. Without complex cache-management logic, the processor hardware is cheaper and easier to design, verify and manufacture. The reduced access energy of simple memories can result in a net energy savings if management overhead is kept low. Software caches can also be customized to each individual program's needs, improving performance or eliminating unpredictable timing for real-time embedded applications. The greatest challenge for a software cache is providing good performance using general-purpose instructions for cache management rather than specially-designed hardware.

This thesis designs and implements a working system (*Flexicache*) on an actual embedded processor and uses it to investigate the strengths and weaknesses of software instruction caches. Although both data and instruction caches can be implemented in software, very different techniques are used to optimize performance; this work focuses exclusively on software instruction caches. The Flexicache system consists of two software components: a static off-line preprocessor to add caching to an application and a dynamic runtime system to manage memory during execution. Key interfaces and optimizations are identified and characterized. The system is evaluated in detail from the standpoints of both performance and energy consumption. The results indicate that software instruction caches can perform comparably to hardware caches in embedded processors. On most benchmarks, the overhead relative to a hardware cache is less than 12% and can be as low as 2.4%. At the same time, the software cache uses up to 6% less energy. This is achieved using a simple, directly-addressed memory and without requiring any complex, specialized hardware structures.

Thesis Supervisor: Anant Agarwal
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

Although my name is the one in large print on the title page, many other people played a significant role in the successful completion of this thesis. First and foremost, I would like to thank my advisor Anant Agarwal for giving me the opportunity to be a part of the Raw group. His leadership has made the Raw group an exciting, rewarding, and enriching place to spend my graduate career. I thank him for his guidance and assistance with my research as well as teaching me how to present my ideas so that the world takes notice. I am also indebted to the other members of my thesis committee, Saman Amarasinghe and Krste Asanović, for their valuable feedback and suggestions.

Next, I would like to thank all the other members of the Raw group, without whom this research would not have been possible. It takes a massive effort to develop and implement an experimental microprocessor and all of the simulators, compilers, libraries, applications, boards, drivers, and other tools needed to make it useful. The list of people who worked on Raw is so long that I'm sure I'll forget someone (and I apologize for that in advance) but I owe it to them all to try. Michael Taylor, Jason Kim, Fataneh Ghodrat, David Wentzlaff and I worked together for many long hours implementing the Raw processor and pushing it through IBM's ASIC flow. Dave, Nathan Shnidman and I spent even longer hours designing, bringing up, and debugging the Raw Single-chip (aka "Handheld") system and its associated FPGA firmware. Dave and I continued on with the design of the Fabric system. I am very grateful to Jonathan Eastep, Albert Sun and Hayden Nelson for finishing what we started by bringing up and debugging the Fabric system in the lab. There are many others that I have had the pleasure of working with including Walter Lee, Ben Greenwald, Henry Hoffmann, Ian Bratt, Theodoros Konstantakopoulos, Jim Psota, Albert Ma, Rodric Rabbah, Csaba Andras Moritz, Ken Steele, Patrick Griffin, Benjamin Walker, and Justin Morin.

I would particularly like to thank Michael Taylor for his excellent work leading the design and implementation of Raw. Less glamorous but just as valuable was the enormous amount of effort he put into the Raw simulator and application development infrastructure. I would also like to thank Matt Frank for the original idea of using software instruction caching on Raw and his valuable guidance when I first started working on this project as a UROP. Volker Strumpen played a crucial role in pushing me to add new features to Flexicache.

Our long discussions on the merits and difficulties of various features helped me to clarify and improve my own ideas. Last but certainly not least, Paul Johnson helped tremendously with the Flexicache preprocessor by developing the binary rewriting infrastructure I used to create it.

On a personal note, I'd like to thank Nathan Shnidman and David Wentzlaff for their generous friendship. See, something good *did* come from the months we spent together pouring over schematics, hunching over a lab bench, staring at a logic analyzer, and checking for over-heating chips with our fingers.

While I'm on the subject of Raw boards, I would be remiss if I did not thank our collaborators at ISI for their hard work and expertise in implementing the Handheld and Fabric systems. We could not have built those systems without the help of Chen Chen, Steve Crago, Matt French, Tam Tho, and Jinwoo Suh.

I'd also like to thank my parents for their many years of support (both financial and emotional!), without which I would not be where I am today. I'd particularly like to thank my father for first introducing me to research and computers (with his spiffy new IBM XT 286) and providing me with the opportunities to pursue my interests in both. I only wish he could be here today to see the results.

Finally, I'd like to thank my wife Krista for putting up with our graduate-student "lifestyle" for so long. I appreciate her sacrifices and support, particularly over the last year, as I've focused all my efforts toward finishing this thesis.

The research presented in this thesis was funded by the DARPA Polymorphic Computing Architectures program, the National Science Foundation and the MIT Oxygen Alliance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Caches have become ubiquitous in modern general-purpose and high-performance microprocessors. By using a small, fast memory to store a subset of the data that is available in a larger, slower memory, they are able to reduce the average data access time and increase performance. Typically, caches are designed so that they function transparently to the software running on the processor. Special-purpose hardware takes care of checking the cache for needed data, fetching data from a larger memory, and managing which subset of the total data is currently held in the cache. However, caches are less common in embedded processors because they increase design and manufacturing costs, use additional energy, and introduce unpredictable delays. Many embedded processors continue to use memory architectures similar to those from the time before caches had been invented.

In the early days of programmable digital computing, caches did not exist. Prior to the introduction of the cache in the late 1960's [94, 58], most computers used a simple two-level memory hierarchy consisting of a fast primary store and a slow secondary store. This hierarchy was exposed and transfers between the two stores were performed explicitly by software [12]. Only the primary store was directly accessible by the functional units; if a program wanted to use instructions or data that were in the secondary store, it had to transfer them to the primary store first. This created the complex problem of deciding what code and data to keep in the primary store for every point in a program's execution [23]. To simplify this task, programmers developed software virtual memory techniques (such as overlays [71, 78] and segmentation [67]) that handled some or all of the memory management tasks automatically. However, as automatically-managed caches became more popular, these techniques fell out of use, were translated into hardware, or were adapted for higher levels of hierarchy for use in operating systems.

Although they increase the complexity of the hardware and require extra energy for every access, caches are popular because they simplify the software and provide a convenient abstraction layer. Specifically, programmers do not need to concern themselves with the size of their code or data: the hardware takes care of mapping a large program into the small on-chip memory. Programmers seeking the ultimate level of performance may still choose to carefully optimize and arrange their code but the cache remains as a backup, guaranteeing correct operation.

However, recent trends in microprocessor design have prompted a second look at the simpler designs of the past. Increasing complexity has led to increased cost and power consumption while producing diminishing returns. As a result, processor designers are looking to reduced complexity and increased parallelism as a road to greater performance. MIT's Raw [84], IBM's Cell [35], and Intel's IXP 2800 [2] are three examples. These designs have replicated, identical processing elements (or *cores*) that each behave as an independent mini-processor. The cores are kept simple to reduce design costs and allow for the maximum number of parallel cores on a single chip. To this end, they use explicitly-managed local memories and exposed memory hierarchies instead of caches. Explicitly-managed memories are simpler than caches and are therefore easier to design, consume less area and energy and permit higher clock frequencies. In addition to these multicore processors, many embedded processors and DSPs continue to forgo caches as well. This is largely motivated by design and manufacturing cost but real-time systems have an additional reason to avoid caches: Caches create unpredictable delays that can cause real-time deadlines to be missed or force programmers to use very conservative delay estimates [9, 57, 16]. Explicitly-managed memories are less expensive and give the programmer complete control.

Although explicitly-managed memories are smaller, cheaper, faster and allow greater control than caches, they are also much more difficult for the programmer to use. Without automatic management, the programmer is responsible for carefully partitioning his program and swapping pieces into local memory as needed. Because this is difficult and time-consuming, the end result will frequently be a coarse-grained partitioning that is inefficient and sub-optimal. Anecdotal evidence suggests that the lack of instruction caches was the most significant programming difficulty with the Intel IXP and other network processors, ultimately leading to their demise.

A software caching system bridges the gap between the hardware and the programmer by providing automatic management of an explicitly-managed memory. All of the traditional cache functions are performed by software which is integrated into the application being executed. Such a system can provide good performance and easy programming without using expensive special-purpose hardware. In addition, because it is implemented in software, it can be easily customized to the specific needs of a particular program. Furthermore, for applications demanding perfect determinism and optimal performance, software caching systems can be omitted, allowing programmers to hand-optimize their programs. Unlike hardware caches that can be disabled and treated like SRAMs, a "disabled" software cache consumes no processor resources whatsoever.

This thesis explores the key components and trade-offs involved in implementing a software instruction cache and presents a complete, working system implemented on the Raw microprocessor.

1.1 Traditional Hardware Caches

To understand the potential advantages and disadvantages of software caches, it is useful to first review traditional hardware caches[1]. The primary goal of any cache is to increase performance by helping bridge the gap between a fast processing core and a (comparatively) slow memory. Although other configurations are possible, the basic example consists of a processor and an external DRAM. A small, fast memory (the cache) is placed between the processor and the large, slow memory and holds a copy of some of the data[2] in the large DRAM. The cache memory is usually located very close to the processor (or integrated into it) so that it can be accessed quickly. When the processor needs a piece of data, the cache is checked first and the data is returned quickly if it is found there (a cache *hit*). If the data is not found (a cache *miss*), a slower access to the DRAM is required. The cache then retains the data retrieved during a miss, in case it is needed again soon.

The basic unit of data manipulated by the cache is called a *cache block* or *cache line*. A cache block (which consists of some number of contiguous bytes in memory) is loaded into

[1]For a more thorough review, see [42, pp 392-448].

[2]Here we use "data" in the most general sense of "information." The information stored in the cache could, in fact, be either program data or instructions.

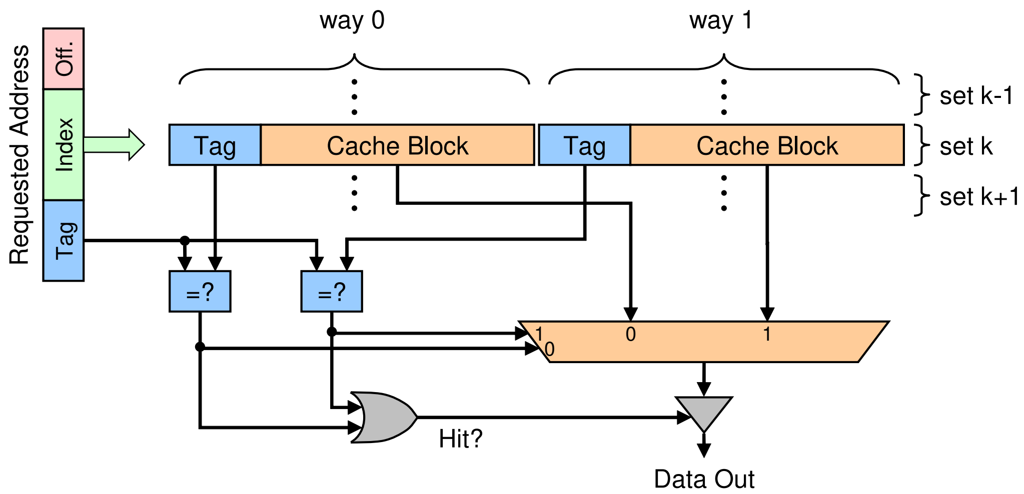


Figure 1-1: Hardware cache operation. A portion of the requested address is used to index to a set in the cache. Tags are stored and retrieved along with each cache line. If the tag stored with one of the ways matches the desired tag, the associated cache block is returned. Otherwise, the required data is fetched from external memory and stored for later use.

or evicted from the cache in its entirety. When the processor needs data from a particular address, the cache checks to see whether the block containing that address is in the cache. If it is, the block is accessed and the requested datum is extracted. If it is not, the entire block (and not just the particular value needed) is fetched from DRAM and stored in the cache for future use.

In addition to the storage required for the actual data held in the cache, a hardware cache needs additional storage and logic to keep track of cache blocks and service requests (Figure 1-1). Because the cache is mapping the large DRAM address space into a smaller memory, several different cache blocks could potentially be stored at each location in the cache. To keep track of which blocks are currently loaded, the cache stores a *tag* along with each cache block. The tag uniquely identifies the original DRAM address of the block. When the cache is accessed, the tag for the desired address is calculated and compared to the tags of the blocks stored in the cache. If the tags of the desired address and a specific block match, the desired block has been found. If no matches are found, a state machine is used to send the request for data to DRAM and handle the response when it arrives.

However, hardware caches typically do not check the tags of *every* block stored in the cache. Instead, they use a hash of the address to select a subset of locations in the cache where the block can be stored. Then, they check the tags of the blocks in only those locations. The subset of locations that are legal for a block is called a *set* and the individual locations within a set are called *ways*. Although there can be any number of ways within a set, most processors use values between one and eight. Instruction caches generally have one or two ways in each set, while data caches typically have two to eight. A cache with n ways in each set is referred to as an *n-way set-associative* cache. However, a 1-way set-associative cache is normally referred to as *direct-mapped* and a cache where any block can be stored in any location is referred to as *fully-associative*.

When a new block is loaded into the cache from DRAM, the cache must select a location in which to store it. On startup, one of the ways within the appropriate set may be empty and can be used immediately. However, in the steady-state, the cache will typically be full and an existing block will need to be *evicted* to make room for the new one. The method that a cache uses to determine which block within a set will be evicted is called its *eviction policy* or *replacement policy*. Because programs tend to have a lot of temporal locality, keeping recently used data in the cache will likely lead to a high hit-rate in the future. Therefore, one of the most widely used policies is to evict the least-recently-used (*LRU*) block in the set. With higher associativities, it becomes more difficult to keep track of exactly which block was least-recently used. Therefore, some caches use pseudo-LRU, FIFO (First-In, First-Out) or Random policies that are easier to implement but still achieve similar results [42, p 400].

1.2 Drawbacks of Hardware Caches

While hardware caches provide fast, convenient access to large memories, they are not without their drawbacks. They are complex subsystems that require substantial effort in initial design, timing closure and verification, thereby increasing time-to-market and development costs. The tags and control logic required for management of the cache consume considerable area and therefore increase manufacturing costs. In addition to area, these structures sit on the critical path for accessing data and can impact timing of the whole processor. Because the cache is implemented in hardware, it is difficult to customize to individual programs' needs. Finally, during operation, caches consume a large fraction of most processors' total power, especially for low-power processors [63, 13]. Much of this power is used for tag checks and control rather than the actual data accesses [99]: an associative tag check consumes considerable energy.

Adding a hardware cache to a processor can be an expensive proposition. Caches (especially highly-associative ones) are complex systems that require large investments of time in the processor design phase. Besides the obvious effort required to design and implement them, they require additional effort to meet processor timing constraints and verify their operation. Because caches are frequently on the critical path in processors, they often require careful tuning and optimization to meet timing constraints. Verification cost is quickly becoming a major problem in the microprocessor industry as designs become more and more complex [10]. Caches contain a large amount of state and require long test vectors to properly test all of it. In addition, some of this state may not be user-accessible (*e.g.*, tags, LRU bits, store buffers) making it difficult to test all corner cases. Long, complex sequences of instructions may be necessary just to get the cache into the state that one wishes to test. Furthermore, caches can impact the debugging of the rest of the processor. Instruction caches are located very early in the processor pipeline and therefore must be working properly before it is even possible to test later stages of the pipeline. Because caches attempt to operate transparently, they can produce different behavior (*e.g.*, hits vs. misses, different blocks evicted) in seemingly similar situations. This sometimes means that a test writer must understand and manipulate the cache, even when testing a different part of the processor (*e.g.*, the external memory interface).

Besides the up-front costs of implementing a cache, there are additional on-going costs such as manufacturing and opportunity costs. Caches take up considerable silicon area in a processor. Therefore, incorporating a cache requires either cutting out other functionality or using a larger die size. Larger dies are more expensive to manufacture, increasing the cost of every processor produced. In the real world, processor designs are subject to production deadlines and budgets so adding a cache probably means giving up something else.

While having a hardware cache clearly provides performance benefits, it is not necessarily an efficient use of space. Only part of the total cache area is used to store the actual data being buffered. A significant amount of area is required for the tags and control logic that are needed to manage that data. Figure 1-2 shows a portion of the layout for the Raw microprocessor that includes a 32 KB, 2-way set-associative data cache. In this example, 29% of the total area of the cache is devoted to tags and control. This area is dedicated solely to caching functionality and cannot be used for anything else. The total cache area

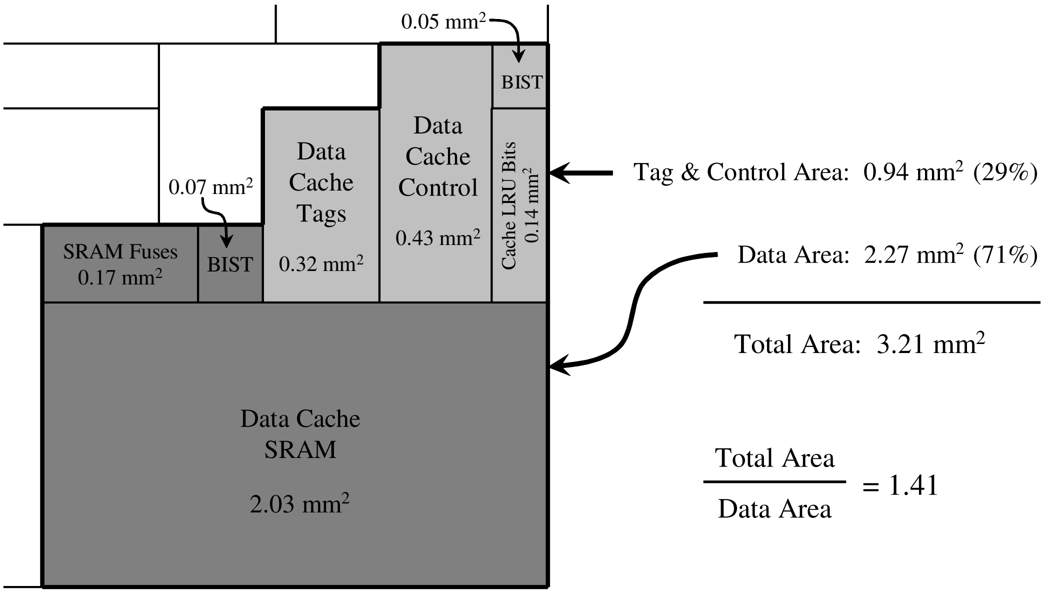


Figure 1-2: Layout of the data cache in a Raw tile. The dark gray area is the portion used for data storage. The light gray area is used for tag storage and control logic. Note that the tag storage and control occupies 29% of the cache area and that the total area is 41% larger than the area needed for data storage alone.

is about 40% larger than the area needed to store the data.

In addition to the area they consume, the tag storage and control logic portions of hardware caches have a propagation delay that can negatively affect overall chip timing. While tag lookup and comparison can sometimes be performed in parallel with the data access, for caches with associativity greater than one, the tag checking logic is frequently the longer path. Data collected from CACTI [95, 76] (a cache modeling tool) indicates that the tag checking path is approximately 20-25% slower than the data access path for a variety of 2-way set-associative caches. Thus the specialized hardware cache structures can become the limiting factor in the rate at which the fetch unit of a processor can be clocked or, alternatively, can require that an extra pipeline stage be added to the fetch unit to meet an aggressive clock goal[3]. Either way, performance can suffer because the clock rate directly affects processor throughput and additional pipeline stages increase branch delay slots or mispredict penalties.

Even when the costs of implementation and production are acceptable, there are other

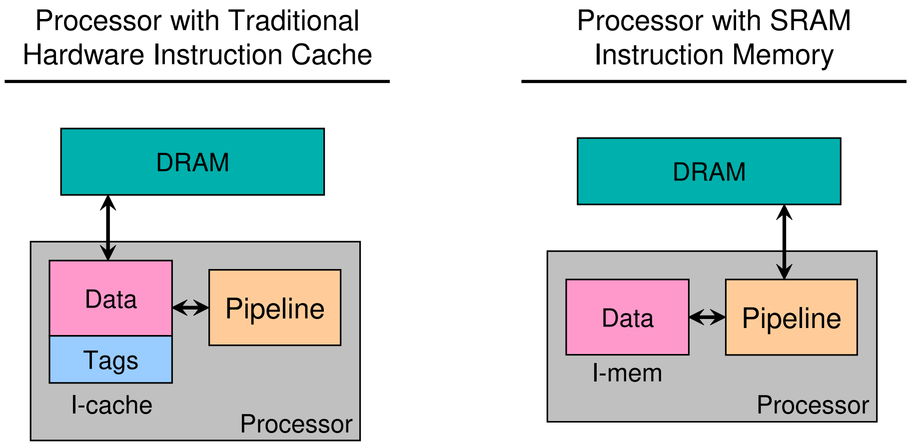
[3]There are way-prediction schemes [68, 48] that allow data to be fetched speculatively and tags to be checked later (off the critical path), thereby hiding the extra latency. However, these schemes require even more area and design effort to implement the prediction logic.

reasons why a hardware cache may not be desirable. Hardware caches operate automatically and in the same way on every program. Because they are implemented in hardware, they usually have little or no ability to be customized for a particular program's needs. As a result, they can produce suboptimal or undesirable behavior for a specific program. As a simple example, consider a program that repeatedly accesses three or four addresses that all map to the same set in the cache. If the cache is direct-mapped or 2-way set-associative, those addresses will conflict and the result will be thrashing (*i.e.*, the same values will be repeatedly evicted and reloaded). A different cache configuration could give better performance but there is seldom a way to modify the cache's behavior. As another example, in real-time environments, caches can introduce unpredictable timing because they are insensitive to the program's access patterns. An unexpected cache miss during a critical routine can result in missed real-time deadlines [66]. This problem is severe enough that most real-time systems avoid caches altogether and accept the extra effort required to use a scratchpad memory [8] that is carefully managed by hand.

Besides affecting performance, caches can also have a large effect on power consumption. Instruction caches, in particular, demand a lot of power because they are accessed for every instruction executed. A survey of power estimation tools [36, 98] and actual processors [63, 13] indicate that the instruction cache is typically responsible for about 25% of the total power consumption of a processor. While some of the energy consumed by the cache is used to perform the required data access, much of it is used to retrieve and compare tags or access data that is not actually needed (see Section 7.2). This is a well-recognized drawback of hardware caches and there have been many proposals that attempt to eliminate some of this overhead using hardware mechanisms [96, 59, 81, 72, 50].

1.3 Simplified Processor Design

The costs and problems associated with hardware caches have led some designers to omit them from their processors. Many embedded processors and DSPs forgo hardware caches in favor of simpler, cheaper alternatives. Even some larger multicore processors such as MIT's Raw [84], IBM's Cell [35], and Intel's IXP 2800 [2] use simple memories to reduce complexity and allow additional functional units to fit on a single chip. Some of these processors (*e.g.*, Texas Instruments TMS470, TMS320C28x, TMS320C000 family, the SPE



(a) (b)

Figure 1-3: Traditional hardware caches vs. simplified memories. In (a), the I-cache automatically manages itself and communicates with DRAM. In (b), software running on the pipeline must explicitly handle all transfers between DRAM and the instruction memory.

of the Cell processor, and the Microengines in the IXP 2800) use architectures similar to the pre-cache machines. They provide only a simple on-chip SRAM to hold all program code and data (Figure 1-3). Additional code and data must be transferred in from external memory before it can be used. Others (such as the TI TMS320C24x and Analog Devices ADSP-21xx families) directly access external memory by default but provide a small on-chip *scratchpad* memory [8] that can optionally be accessed instead. Other embedded processors with similar architectures include the Analog Devices ADSP-21160m and ADSP-TS201S; Atmel AT91-C140; ARM 968E-S; Hitachi M32R-32192 and SuperH-SH7050; Infineon XC166; and Motorola/Freescale MPC500, Coldfire 5206E, and Dragonball.

Both of these types of on-chip memories are simple array structures that are directly-addressed and explicitly managed by software. Compared to hardware caches of the same data capacity, they are smaller, have shorter access times and consume less energy per access. For example, a 32 KB SRAM memory is about 30-40% smaller (Figure 1-2 and [8]) and uses 20-50% less energy (Table 7.4 and [8]) than a 32 KB, 2-way set-associative cache. They are also more predictable than caches since the programmer can control whether something is present in the memory. On the other hand, these simple memories are much more difficult for the programmer to use. Since they are not automatically managed, the programmer typically needs to painstakingly partition his code or data into manageable pieces and then manually copy the pieces into or out of the local memory as needed. This requires considerable effort and a detailed level of knowledge about the program that is difficult to obtain when using high-level languages.

1.4 Software Caching Overview

To ease this programming burden and fill the role played by hardware in a traditional cache implementation, we propose implementing the same functionality in a software system. Obviously, the processor must still contain a small, fast memory to store the information being cached. However, all the other functionality traditionally performed in hardware is instead performed by software running on the general-purpose processing core. This software is integrated into the user application so that when a program is loaded and run, it automatically begins managing the local memory as a cache. The management functions include 1) determining how the local memory will be organized, 2) keeping track of the data that has been loaded, 3) receiving or intercepting requests for data, 4) checking to see if needed data is present, 5) fetching data from external memory if it is not, and 6) deciding how and where to store that data. Although these are essentially the same tasks that were performed by the old virtual memory systems, a different approach to the problem is taken here by applying modern code-caching techniques to modern architectures.

One of the key goals of a software caching system is the *automatic* management of the local memory. Thus it is important that the system be as transparent as possible to the application programmer, so that he need not be intimately familiar with the inner workings of the cache. Nor should he have to make extensive changes to the way he writes programs to make them compatible with the caching system. Ideally, the programmer should be able to ignore the caching system completely, just as he would with a hardware cache. The application program should be written as if the main system memory (typically external DRAM) is the only memory; the caching system will take care of any modifications needed to actually run the program using the processor's local memory.

However, the fact that the caching system is integrated into the user program means that there may be some situations where perfect transparency cannot be preserved. In addition, there may be situations where the programmer does not want total transparency. A sophisticated programmer may want to exercise some control over the caching system to gain greater performance, eliminate timing uncertainty or optimize for other specific needs.

Therefore, the caching system should provide interfaces that allow this control. A software caching system provides a flexible abstraction boundary. The programmer can respect the boundary for convenience or choose to burrow through it for increased optimization.

1.4.1 Potential Benefits

Implementing a cache as a software system has the potential to address all of the drawbacks to hardware caches that were mentioned above. A simple memory (like the SRAM or scratchpad memories used by the aforementioned embedded processors), is cheaper and easier to design and manufacture. All of the cache management hardware (including tag storage, tag comparison, LRU bits, store buffers, cache miss handler, etc.) is eliminated. All that remains is the data storage memory and a very small amount of control logic. This control logic is required to implement new load and store instructions in the processor's ISA that directly access the data memory. These instructions do not do any address translation: the data memory is treated as its own independent address space.

This type of architecture is substantially easier to implement than a hardware cache. This is evident in the Raw microprocessor which contains an SRAM instruction memory and a hardware data cache. The data cache requires approximately 2000 lines of Verilog code to implement while the instruction memory requires less than 100. Similar differences were observed in the amount of time spent writing test vectors and the number of bugs discovered in each unit. Furthermore, the data cache actually had to be thrown out and completely redesigned at one point to improve performance and timing.

The instruction memory also requires much less chip real estate than the data cache. Even though both memories have the same data capacity, the data cache occupies 1.4× the area that the instruction memory uses. When designing a system with an explicitly-managed memory, the area saved by eliminating the specialized cache hardware can be used either to decrease manufacturing costs or to increase performance. If cost is the primary concern, the smaller footprint may allow the use of a smaller die. If performance is paramount, the additional area can be used to increase the size of the data storage or add additional functional units.

Accessing a simple memory can also be faster than accessing a hardware cache. Assessing the impact of using a cache versus a simple memory on the overall timing of a processor is difficult to do without evaluating two complete designs. However, in Raw, the data cache occupies three pipeline stages while the instruction memory fits in one. The critical path in Raw is in the fetch unit (which includes the instruction memory) so any additional delay would certainly have required a slower clock frequency or an extra pipeline stage.

Since the cache operation and control is now a part of the user program (rather than permanently set in hardware), it can be customized to each individual application. Based on programmer knowledge, static analysis or profiling, adjustments can be made to guarantee real-time deadlines, avoid pathological cases or optimize for different goals. For example, one program might be optimized to save energy by minimizing cache misses (and therefore power-hungry DRAM accesses) while another might be willing to accept a higher miss rate in exchange for simplified control and thus, higher performance. In fact, different portions of the same program can even be treated differently. Part of the instruction memory can be dedicated to keeping performance-critical code resident while the remainder is used as a cache for the rest of the program. Of course, the ultimate pay-off occurs when a program is small enough to fit entirely in the local memory. In this case, the caching system can be omitted entirely and the program can reap all the benefits of the simplified hardware without any additional overhead. Because the caching system is implemented in software, this decision can be made on a program-by-program basis, providing only the functionality that is needed.

A software cache system has the potential to consume less energy than a traditional hardware cache. Compared to a hardware cache, a simple SRAM memory requires much less energy for each individual access. This is because the simple memory performs only a single data access while the hardware cache must also access the tag storage and compare the desired tags to the ones it retrieves. In addition, caches with associativity greater than one typically access all of the ways within a set simultaneously but wind up discarding the data from all the ways whose tags are incorrect. Therefore, the actual data accesses or instruction fetches in a software cache will consume less energy. However, a software cache must execute additional instructions to service requests and manage itself. These extra instructions will consume additional energy and counteract the previous gains but, as long as they are not excessive, the total energy consumed by the processor can still be reduced.

In addition to processor energy, a software cache may be able to reduce the total system energy consumption by reducing the number of DRAM accesses. This reduction can be due to either customization to a specific program's needs or the use of more sophisticated management policies that would not be practical to implement in hardware. Since DRAM accesses can consume one to two orders of magnitude more energy than on-chip memory accesses or additional instructions [43], the total system energy consumption may be reduced, even if the software cache increases the processor energy consumption.

1.4.2 Challenges

Even though software caches have the potential to solve many of the problems associated with hardware caches, there are several challenges to overcome before this potential can be realized. One challenge, maintaining transparency, has already been discussed. Another is making efficient use of limited local storage resources. A balance must be struck between space used for management and bookkeeping and space used to store cached information. Perhaps the greatest challenge is providing good performance. Not only is good performance important in its own right, it also improves energy utilization by reducing the number of instructions executed. It is not difficult to design a system that provides better performance than one that has no cache at all and goes directly to external memory for every access. However, a processor with no cache is not really the appropriate competition. To be a viable option in real systems, a software cache needs to be competitive with hardware caches and hand-coded explicitly-managed systems.

Unfortunately, the software system starts at a disadvantage. Hardware caches are designed with special-purpose logic that performs precisely the required operation. Software caches, on the other hand, may have to execute many general-purpose instructions to achieve the same result. Even worse, the hardware can be designed to perform many such operations in parallel while the software is generally limited to a single sequential thread. As a result, all of the management operations of the cache (checking for a cache hit, sending out requests on a miss, updating bookkeeping data structures, etc.) will take longer with software, especially if the operation of a hardware cache is closely imitated. Instead, the software cache must employ mechanisms that are uniquely suited to its strengths and attempt to optimize away expensive operations whenever possible.

1.4.3 The Flexicache System

The previous discussion of software caches applies equally to both instruction and data caches. However, in this thesis, we focus exclusively on the problem of implementing an *instruction* cache using software. Although instruction and data caches are usually implemented very similarly in hardware, it does not necessarily follow that they should be implemented similarly in software. The two types of caches are accessed in very different manners and patterns. Instruction caches are accessed implicitly for every instruction whereas data caches are accessed only by instructions that explicitly request memory access. Program code is easily analyzable and highly predictable. Most instructions do not affect the flow of control so the fetches are sequential. Most instructions that *do* affect control-flow either jump to a single destination or choose between two possible destinations. Therefore, at each point in the program, there is usually a very short list of addresses that may need to be fetched next. In contrast, data accesses can be much more complicated. A single instruction can access many different locations on successive executions and neighboring memory accesses do not necessarily have any relation to each other. Because of these differences, different techniques are required to analyze and optimize accesses. For more information on software *data* caches, see Chapter 9.

Flexicache is a software instruction-caching system that we have implemented to better understand the key components and design choices involved in a software instruction cache. This system is designed as a tool to help a developer prepare his application for use on a processor with an explicitly-managed instruction memory. First, the programmer writes his application just as he would for a processor with a hardware instruction cache. Then, the software caching system is applied to the binary file and becomes integrated into the executable. When the application is run, the Flexicache system automatically assumes control of the local instruction memory and begins managing it as a cache.

The Flexicache system is composed of two pieces: a preprocessor and a runtime system. The preprocessor (implemented as a binary rewriter) takes the original application binary and translates it into an I-cached binary. This involves breaking the program up into blocks and redirecting control-flow instructions to the runtime system. The runtime system is attached to the translated binary and is automatically loaded and run when the application is executed. It remains resident in the instruction memory and handles requests from the running application for new blocks of code. (A detailed diagram showing the entire workflow is given in Chapter 2.) When the modified control-flow instructions jump to the runtime system, it checks its internal data structures to see if the code that is needed next is already present in the cache. If not, the correct block is loaded from DRAM and stored in the instruction memory, just as it would be with a hardware cache. Now that the needed block is in the cache, the runtime system jumps to it and the cycle repeats.

This combination of a static off-line preprocessor and a dynamic runtime system provides an excellent balance between efficiency and adaptability. The preprocessor does as much work as possible ahead of program execution time and is able to use complex, expensive analysis because it is done off-line. The runtime system handles events that cannot be predicted statically (like the direction of a branch) and can be streamlined since the program translation is already done. This allows the Flexicache system to adapt to different program phases or workloads while keeping runtime overhead low. In addition, different applications can use different preprocessors or runtime systems, allowing Flexicache to be customized for each application.

1.5 Thesis Contributions

This thesis investigates the feasibility, strengths, weaknesses and trade-offs of implementing a level-one instruction cache as a software system. It makes the following contributions:

- The first work to use modern code caching techniques to implement an automatically-managed, level-one instruction cache *entirely* in software. Our system runs entirely on the core CPU, using only a simple SRAM instruction memory, and does not rely on co-processors, hardware support for cache hits, or other complex logic structures. To achieve good performance, we use a novel hybrid static-dynamic approach which incorporates a static, offline preprocessor and a dynamic runtime system.

- Demonstration of a complete, practical software I-caching system implemented on an actual hardware platform (the Raw microprocessor, see Appendix A). Our system is independent of the source programming language and provides support for interrupts, pinning of arbitrary functions in the cache and some self-modifying code. It is robust enough to be used as a standard tool by other researchers using the Raw microprocessor.

- Identification of the crucial components and user interfaces required for such software I-caching system. This particularly includes areas where it is difficult, impossible or undesirable to maintain complete transparency. Examples include code pinning for predictable timing and support for self-modifying code. While these issues may not arise for simple applications in simulated environments, they become crucial when supporting large, complex applications on real hardware.

- Identification of key factors that may negatively impact performance and several optimizations that address them. These optimizations focus primarily on eliminating explicit checks for the presence of code whenever possible. We introduce two novel optimization techniques: *macroblock fusion* and compound instruction decomposition. *Macroblock fusion* combines the benefits of low-overhead fixed-size cache blocks and compact variable-size cache blocks. *Compound instruction decomposition* deals with hard-to-optimize compound processor instructions by separating them into multiple simpler operations.

- Identification of several hardware architectural features that can improve the performance and efficiency of a software instruction-caching system. The proposed features are much simpler and cheaper to implement than a full hardware cache but allow a software cache to achieve comparable performance.

- Evaluation of the performance and energy consumption of the Flexicache system on standardized benchmarks, including the results of adjusting system parameters and the impact of various optimizations.

Our results show that software instruction caching can be a viable option for processors with explicitly-managed memories. Flexicache is easy to use and provides excellent transparency. Most programs require no modifications to use it. Only special cases like self-modifying code and the use of interrupts require any cooperation from the programmer. The system is robust and complete enough that it is used as a standard tool by other researchers working on Raw. It allows them to run large programs using the small local memory on each tile without having to concern themselves with manually orchestrating transfers from DRAM. Performance is good on a variety of benchmarks. In most cases, the software cache incurs between 2.4% and 12% overhead versus a similarly sized hardware cache. At the same time, the elimination of tag checks and fetches of unused ways allows the software cache to consume up to 6% less total energy than the hardware cache. This thesis forms a solid foundation for future work in the area of software caches. Additional improvements are possible through the use of further optimizations, alternative techniques and light-weight hardware support.

1.6 Chapter Summary

This chapter introduced the concepts and potential advantages of a software instruction-caching system. A brief overview of traditional hardware caches was given to review basic caching concepts and terminology. The goals and concepts of a software caching system were then described and the potential advantages versus a hardware cache were discussed. The key advantages of software caches are their flexibility, ability to be customized to individual applications, and reduced implementation costs. The key challenges to building an effective software caching system are maintaining a clean abstraction layer for the programmer and providing good performance using general (rather than specialized) hardware resources. Next, the Flexicache system was introduced and its two components (a preprocessor and runtime system) were briefly described. Finally, the major contributions of this thesis were highlighted. These include: a pioneering implementation of a software instruction-caching system, a detailed description and analysis of this system, and ideas for improving the system with future hardware and software mechanisms.

1.7 Thesis Organization

The remainder of this thesis is structured as follows: Chapter 2 provides an overview of the Flexicache system, including key features, techniques, and basic operation. Chapter 3 contains a brief overview of the Raw microprocessor, focusing on the specific features relevant to Flexicache. Chapter 4 describes the internal workings of our baseline Flexicache implementation for Raw in more detail. Chapter 5 discusses several features that are important for real applications where complete transparency cannot be maintained. To allow the programmer to benefit from these features, interfaces into the Flexicache system are defined. Chapter 6 describes several important optimizations and improvements to the baseline system.

Chapter 7 evaluates Flexicache from several viewpoints including performance and energy consumption. Results are presented from both Mediabench and SPEC benchmark suites. Chapter 8 identifies several small hardware features that can be used to improve the performance of a software instruction-caching system. Some of these features are found in Raw while others are suggestions for future processor designers that wish to provide support for software instruction caching.

Chapter 9 outlines the wide variety of related work and systems that form a foundation for this work. Chapter 10 discusses future directions for Flexicache including additional optimizations and alternative designs. Chapter 11 concludes by summarizing our findings and discussing lessons learned.

Appendix A contains additional background information on the Raw microprocessor including detailed descriptions of all features relevant to Flexicache. It also describes two computer systems built around Raw that were the target platforms for our implementation. Appendix B provides flowcharts showing the operation of the Flexicache preprocessor and runtime system.

Chapter 2

Flexicache System Architecture

Flexicache is a software system we have designed and implemented to explore the concept and trade-offs of software instruction caching. This chapter introduces the Flexicache system and describes its key components and basic operation. Additional implementation details will be given in Chapter 4. Optimizations to this core implementation will be discussed in Chapter 6. The goal for Flexicache was to create a complete, practical solution that would allow users to run programs of arbitrary size and complexity on processors with explicitly-managed instruction memories. The system should provide good performance and maintain transparency to the greatest extent possible. In other words, the programmer should be able to write his program as if there is only a single large memory yet receive the performance benefits of a small, fast instruction memory. However, the programmer should also have the option of exercising some control over the I-caching system to optimize for his particular needs.

2.1 Processor Hardware Model

For this thesis, we assumed a processor architecture similar to what might be found in a modern embedded processor with an explicitly-managed instruction memory. Figure 2-1 shows an overview of the target architecture. It consists of a conventional, in-order, single-issue pipeline with separate instruction and data memories. A single-issue, in-order pipeline is assumed because this is the most likely design for a low-cost or small-footprint processor. However, our techniques will also work with more complex pipelines. The data memory could be either an explicitly-managed memory or a traditional cache: the Flexicache system

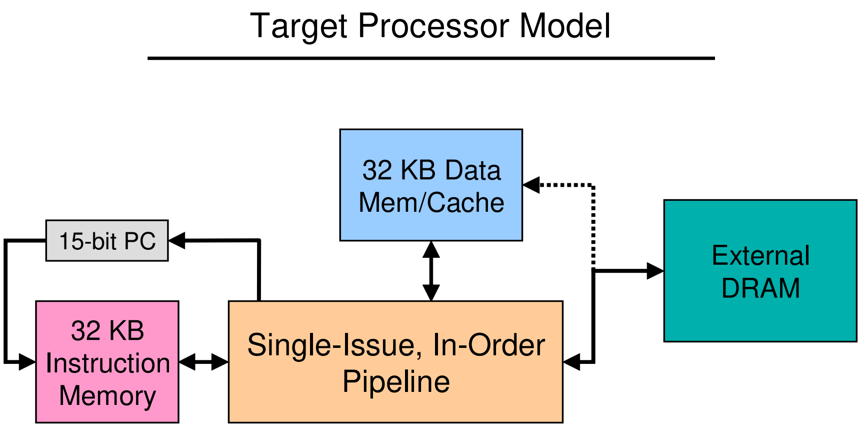


Figure 2-1: The target processor model consists of a conventional RISC pipeline with separate instruction and data memories. The pipeline can write to the I-mem and directly access DRAM to retrieve blocks of code. The program counter (PC) directly indexes into the I-mem, requiring all code to be stored there before it can be executed. The data memory could be either an explicitly-managed SRAM or a cache.

deals only with the instruction memory and is, therefore, independent of the data memory architecture. In fact, Flexicache could also be used on a system with a single unified memory as long as it was statically partitioned between instruction and data usage.

On this processor, all instructions must be resident in the instruction memory (or *I-mem*) before they can be fetched and executed by the pipeline. Instructions stored in DRAM must be explicitly copied into the I-mem (either via special instructions or a DMA engine like the one in the Cell processor's SPE [35]) before they can be run. This simplifies the instruction fetch logic (versus allowing instructions to be run from both DRAM and the I-mem) and is the norm for higher-performance processors. This type of architecture is likely to become even more common among embedded processors as the performance gap between processors and DRAM grows. As a side effect of this requirement, the processor's program counter (PC) and the destination fields of all control-flow instructions only need to have enough bits to address the relatively small I-mem.

One very important requirement for the target processor is that it be able to efficiently read and write instructions in the I-mem. This feature is used to copy code from DRAM to the I-mem but is also crucial to the success of most of our optimizations (which are presented in Chapter 6). In addition, it allows a portion of the I-mem to be used to store runtime data structures. All that is needed to meet this requirement is two relatively simple instructions: 1) load the value from an I-mem location and 2) store a value to an I-mem location. The I-mem should be addressable at the granularity of a single instruction word.

Finally, a note on terminology: To distinguish between the I-mem and DRAM address spaces, we adopt the terms *physical address* and *virtual address* from the virtual memory community (see related work in Section 9.1). Addresses within the I-mem are referred to as *physical* addresses since they are the addresses understood directly by the processor hardware. Addresses in the external memory are referred to as *virtual* addresses since they form the larger program address space that is mapped into the I-mem by the software I-caching system.

2.2 Flexicache Overview

The Flexicache system consists of two components: a preprocessing program and a runtime system. The preprocessor analyzes the original user program (which assumes a large address space) and then modifies it to use the runtime system. The runtime system manages the cache during program execution. Figure 2-2 shows how the Flexicache system fits into the application compilation and execution toolchain.

All but the simplest programs make use of data and instructions in a dynamic way that is impossible to fully predict before execution. Therefore, a software caching system must have a runtime component to handle these dynamic events. The runtime system is responsible for servicing requests for instructions and managing the I-mem. In our case the runtime system is written using normal processor instructions and executes on the same processor as the user program. It remains resident in the lower part of the I-mem and uses the upper part for storing blocks retrieved from DRAM. The runtime system keeps track of what code is currently in the I-mem, fetches code from external memory when needed, decides where to store that code when it arrives, and evicts code when necessary.

However, any work that can be done off-line reduces the amount of work that needs to be done at runtime. Therefore, any efficient software caching system will likely have an off-line preprocessing phase as well. The preprocessor has two primary tasks: dividing up the program into convenient chunks and modifying instructions to make calls to the runtime system when new code is needed.

After preprocessing, the modified user program is linked with the runtime system to create the final, cache-enabled binary. As a result, a portion of the I-cache system is

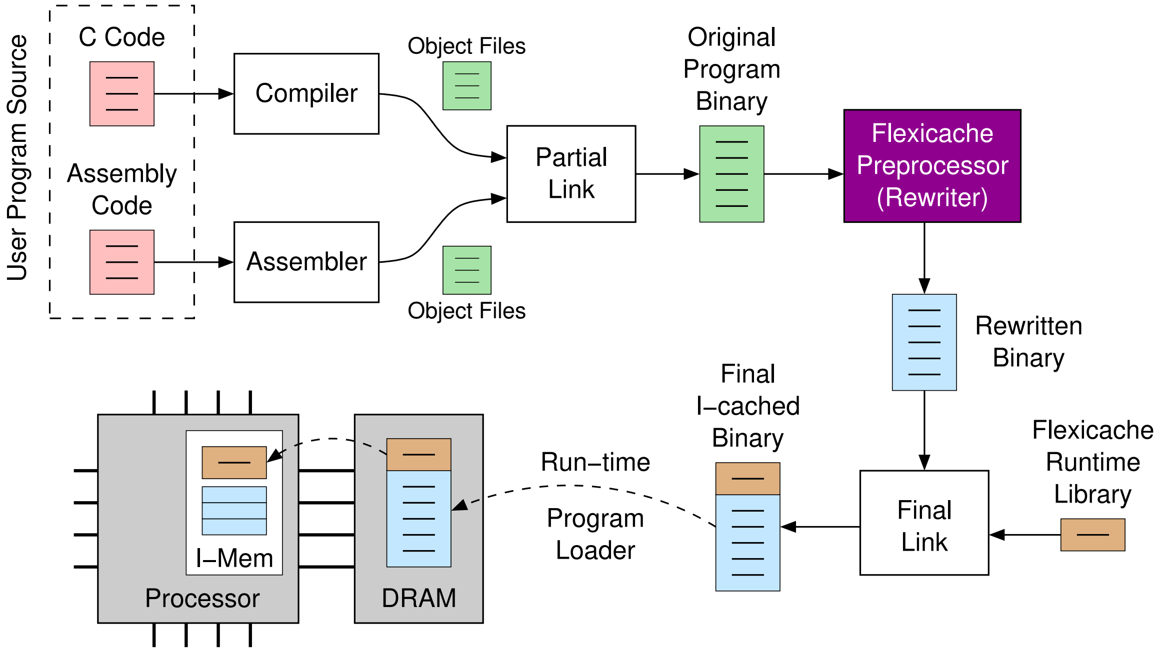


Figure 2-2: Flexicache System Overview. The user's program is compiled and linked into a single object file. A preprocessor modifies it for I-caching and it is linked with the runtime system library. At runtime, the complete binary is loaded into DRAM and the runtime system is copied to I-mem. The runtime system then fetches blocks from DRAM as needed. The preprocessor and runtime library make up the software I-caching system.

actually integrated into each program. This means that the caching system can be fine-tuned to produce the best results for each individual program. Alternatively, a developer could produce different versions of his program that were optimized for different needs. For example, a handheld device might use a low-power version of the program when running on batteries and a high-performance version when plugged into the wall. Using the same basic hardware, different programs could potentially use completely different caching schemes. The system we have developed is one option but alternative designs are certainly possible. As a trivial example, consider a very small program that will fit into the I-mem in its entirety. This program could omit a caching system altogether and achieve optimal performance and energy consumption.

The basic unit of code manipulated by the cache is referred to as a *cache block*. The cache is managed by pulling entire blocks of code in from external memory as they are needed. As long as execution stays within a block, the next instruction is guaranteed to be present in the cache. When the flow of control leaves a block and moves to a new block, the runtime system must check its data structures to determine if the new block has already been loaded. If it has, control is transferred to the new block immediately. If it has not, the runtime system retrieves the block from external memory, copies it into the cache (evicting an older block if necessary), updates its data structures and then transfers control.

2.2.1 User Program Modification (The Flexicache Rewriter)

One of the key goals for Flexicache was to preserve transparency for the programmer to the greatest extent possible. The programmer should be able to write his program as if there is only a single large memory space. However, this goal does not extend to the application binary itself. Since the underlying hardware does not provide an abstraction layer to hide the memory hierarchy, the software executing on the processor *must* be aware of it. Hence, there must be a modification pass introduced at some point to translate the user's original program into an I-cached program.

This modification may take place at any one of a number of points in the tool chain. Potential choices include: the final stage of the compiler, before or during linking, during loading of the program or even during execution. In Flexicache, the code modification pass is performed by a binary rewriter that operates on object files just before the linking stage. Using a binary rewriter allows us to add instruction caching to programs from a number of different sources without having to write a code modification pass for each source compiler. It also gives us the potential to take extra time and memory to perform a more complex analysis than would be practical with a loader implementation. One disadvantage of using a pre-link rewriter is that the final binary is locked into a specific version of the I-cache system. A load-time implementation would allow a single binary to be distributed (possibly with extended symbolic information to aid rewriting) and then customized for a particular goal or processor each time it is run. Although we have chosen a pre-link implementation, there is nothing fundamental to our technique that requires this. The same end result could have been achieved using a preprocessor operating at any stage.

The rewriter performs two primary tasks to modify a program for software instruction caching as shown in Figure 2-3. The first task is to divide up the original instructions in the user's program into the cache blocks that will be managed by the runtime system later. The second task is to examine each block and introduce a call to the runtime system each time the flow of control would leave that block.

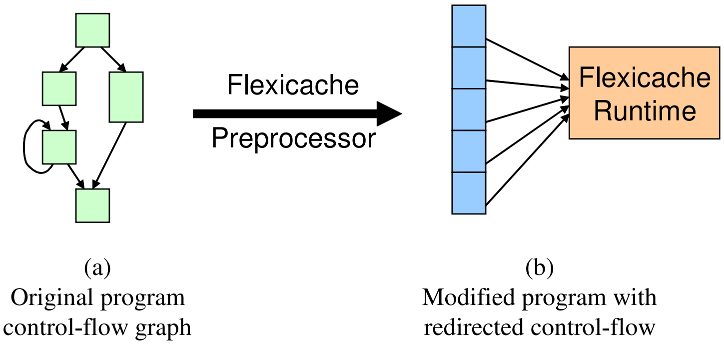


Figure 2-3: Flexicache preprocessor tasks. The preprocessor performs two tasks: dividing the original program up into cache blocks and redirecting control-flow paths that leave the cache blocks to the runtime system.

Cache Block Formation

A *cache block* is defined as a sequence of program instructions that are retrieved, stored and evicted as a single unit. It is often referred to as a "cache line" in a hardware cache. There are many possible ways to form blocks from the original program. Hardware caches simply chop up the program into fixed-size lines without regard for the program's structure or control-flow. While this approach could be used in a software cache, the use of an off-line preprocessor provides an opportunity to analyze the program and take advantage of its inherent structure. The standard way of looking at a program's structure is by dividing it up into a collection of *basic blocks* [5, p 528]. Basic blocks are single-entry, single-exit sequences of instructions with no internal branches.

Basic blocks provide a very convenient partitioning of a program for a software instruction cache because they are bounded by potential changes in control-flow. Because they execute sequentially from beginning to end, there is no uncertainty about which instruction will be required next until the end of the block is reached. Thus, if a basic block is loaded into the cache in its entirety, the body of the block may execute without any runtime checks to determine if the next instruction that is needed is already present. Only at the end of the block, where flow of control transfers to a new block, is a runtime check required. In addition, a basic block represents the largest sequential series of instructions that are guaranteed to execute atomically, *i.e.*, when any of them is executed, they will all be executed. If any larger series of instructions is considered, some of those instructions may not be used on every invocation. Thus, if the cache loads one basic block at a time, only instructions that will actually be executed will be fetched.

Control Flow Modification

The key to the operation of the software instruction cache is the redirection of control-flow paths that might leave the current cache block. These paths must be modified to make a call to the runtime system so that it can determine if the intended destination block has already been loaded and jump to that block's location if it has. (Although it is referred to as a *call* to the runtime system, it is not a procedure call in the traditional sense since the runtime system will not be returning back to the code that called it.) Paths leaving a cache block can be created either by control-flow instructions (*i.e.*, branches and jumps) or by simply falling through to the next block.

2.2.2 Flexicache Runtime System

The runtime system is the heart of the software instruction caching system. It is responsible for managing the I-mem as a cache during program execution. The runtime system is essentially a library written in assembly language for speed and ease of direct hardware interaction. After the user program is preprocessed, it is linked with the runtime code to form the complete final binary.

When it comes time to execute the program, the program loader loads the entire binary into DRAM and then copies the first part (which includes the Flexicache runtime system) into the lower part of the I-mem, as illustrated in Figure 2-2. The runtime system will remain resident in I-mem and use the remaining space to store cache blocks. The program loader then transfers control to the runtime system's initialization routine which loads the first cache block of the user program into the I-mem and transfers control to it.

Figure 2-4 illustrates the operation of the runtime system. As the user program executes, the control-flow paths modified by the rewriter will create calls into the runtime system whenever it is possible that the next instructions needed are not yet in the cache. When the runtime system receives one of these requests for code, it checks its internal data structures to determine if the needed code is currently present in the I-mem. If it is, execution can immediately continue with that code. If not, the miss handler is invoked to fetch the correct cache block from DRAM and store it in the I-mem so that it can be executed. Once the internal data structures have been updated, control is passed back to the user program and the cycle repeats.

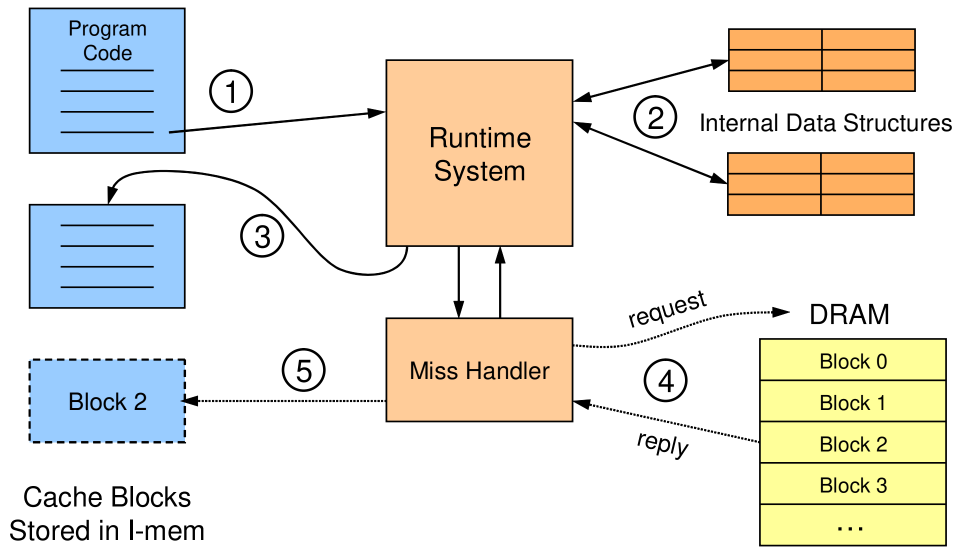


Figure 2-4: Flexicache runtime overview. As the user program executes, it will encounter one of the control-flow instructions modified by the preprocessor (1). This causes execution to transfer to the runtime system which checks its data structures (2) to see if the next block is already in the cache. If so, execution can continue (3). If not, the miss handler retrieves the necessary block from DRAM (4) and stores it in the I-mem (5). After updating the data structures, execution continues with the new block and the cycle repeats.

2.2.3 Implementation Issues

The basic architecture and operation of the Flexicache system is fairly simple; however, as with many real systems, the key to developing a practical, useful system is getting the details right. Many of the important decisions and trade-offs in such a system do not become apparent until a detailed implementation is created. In addition, some of the greatest challenges in a software system may be overlooked if it is not used on actual computer hardware to run real applications.

The subsequent chapters describe the complete implementation of the Flexicache system that we have developed for the Raw microprocessor. This implementation runs on both the Raw simulator and the actual prototype hardware systems that we have built using Raw (see Appendix A). It has been evaluated using a variety of applications from two different industry-standard benchmark suites. However, it has also been used by other researchers in the Raw group as a standard part of the application development toolchain. Support for features such as pinned code, interrupts, and self-modifying code might never have been included were it not for these users demanding them. In the end, these have turned out to be some of the most challenging (and interesting) features of the Flexicache system.

2.3 Chapter Summary

This chapter introduced the Flexicache software instruction-caching system. The system is composed of two parts: a preprocessor (the *binary rewriter*) and the runtime system. The rewriter prepares the original uncached program for caching by breaking it into cache blocks and modifying control-flow instructions to make calls to the runtime system. The runtime system manages the I-mem as a cache during program execution by fetching and storing program cache blocks as they are requested by the running user program.

Chapter 3

Raw Microprocessor Overview

This chapter provides a very brief overview of the Raw microprocessor. To explore software instruction caching in greater depth, we have implemented a complete version of the Flexicache system for the Raw microprocessor. This chapter highlights the specific features of Raw that are mentioned in the detailed description of our implementation found in the next several chapters. While the Flexicache system approach will work on many different processors with explicitly-managed memories, it is helpful to have a specific example in mind when describing different features and mechanisms. A more detailed description of the Raw processor and two computer systems that were built around it can be found in Appendix A.

3.1 High-Level Architecture

The Raw microprocessor is a member of a new generation of tiled multicore processors. A Raw processor is composed of a 2-D array of identical *tiles* as shown in Figure 3-1. Each tile contains a simple computational core and two types of communication routers (*static* and *dynamic*) that connect it to the neighboring tiles. The computational core in each tile consists of a basic RISC processing pipeline with separate instruction and data memories and closely resembles the abstract processor model presented in Section 2.1. Since each tile has its own independent instruction memory and program counter, our implementation of the Flexicache system for Raw treats each tile as a separate processor. This section highlights the key features of Raw that are relevant to the Flexicache implementation.

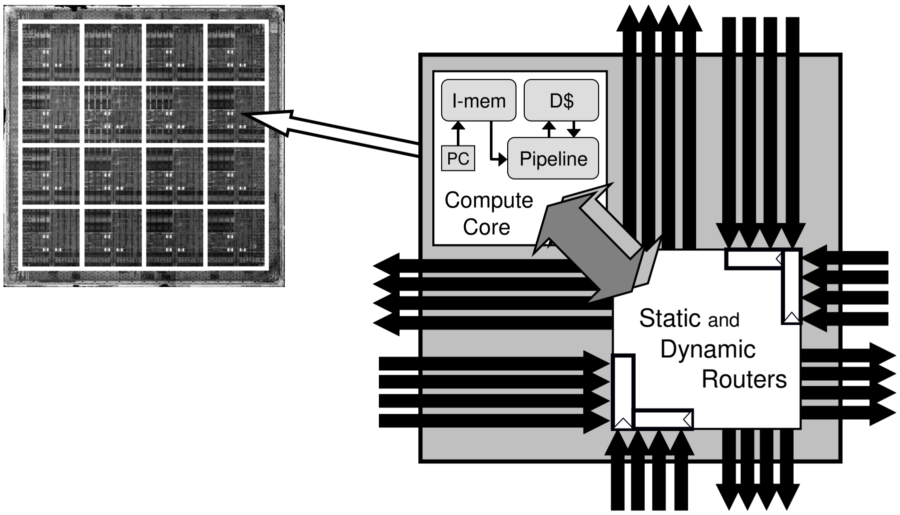


Figure 3-1: Overview of the Raw architecture. A Raw processor is composed of a 2-D array of identical tiles. Each tile contains a basic 32-bit computational core and two types of routers to connect it to the neighboring tiles.

3.2 Instruction Memory Architecture

The computational core in each Raw tile is essentially a simple 32-bit RISC processor supporting a MIPS-style ISA. It consists of a conventional 8-stage, single-issue, in-order pipeline with a 32 KB data cache and a 32 KB explicitly-managed instruction memory. The instruction memory (or *I-mem*) is located in the first stage of the pipeline and is directly accessed for instruction fetches. All program instructions must be copied into the I-mem before they can be executed. This is accomplished using a special `isw` (*I-mem Store Word*) instruction that stores the value from a register into a specified location in the I-mem. There is also a corresponding `ilw` instruction that allows a value to be read out of the I-mem and placed in a register. Together, these instructions allow the contents of the I-mem to be arbitrarily manipulated at the granularity of a single instruction.

3.3 Interrupts

Interrupts on Raw are local to each tile and operate in a fairly conventional manner. Interrupt mechanisms are relevant to a software instruction-caching system because they involve a change in control-flow and may require a handler routine to be fetched from DRAM. There are two different categories of interrupts on Raw (*system* and *user*) with system-level interrupts having higher priority.

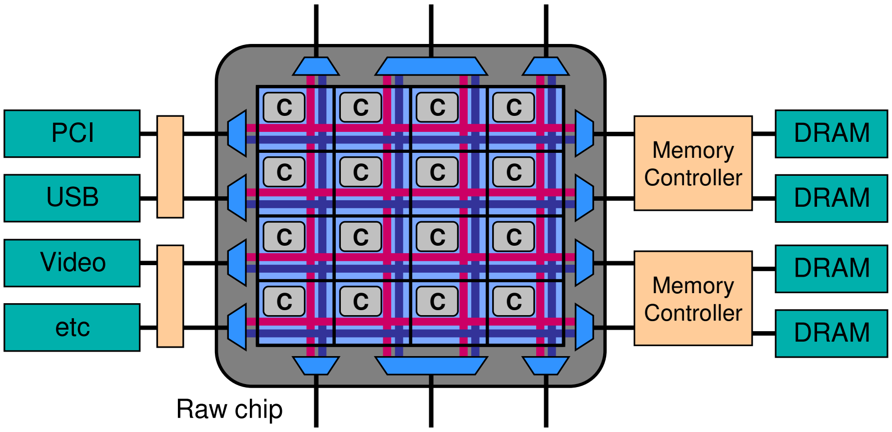


Figure 3-2: Architecture of a simple Raw-processor-based system. On-chip communication networks connect Raw tiles together and extend off-chip to form general I/O interfaces. DRAM controllers and other I/O devices can be attached to these interfaces to form a complete Raw system.

When an interrupt fires, future interrupts are disabled, the resume address is saved, and the corresponding interrupt handler is invoked. The resume address is placed in a dedicated status register: EX_PC if the interrupt is system-level and EX_UPC if it is user-level. The interrupt vector table in a Raw tile is slightly unconventional in that it stores four instructions for each interrupt rather than a pointer to a handler routine. When an interrupt occurs, the processor begins fetching and executing the instructions from the appropriate table entry. In the simplest case, these instructions will just jump to the actual interrupt handler. However, since there is space for four instructions, there is room to perform a slightly more complicated operation if needed. When the interrupt handler is finished, it ends with an `eret` (for a system-level interrupt) or `dret` (for a user-level interrupt) instruction. These instructions atomically jump to the address stored in EX_PC or EX_UPC and re-enable the appropriate level of interrupts.

3.4 On-Chip and Off-Chip Communications

The computational core in each tile is connected to the other tiles and off-chip I/O resources via on-chip static and dynamic networks. These networks form a mesh across the entire area of the chip and connecting all tiles to each other. They also extend off-chip at the periphery of the die, creating a general I/O interface. Figure 3-2 shows how I/O devices are attached to Raw to form a complete system.

The static networks are used primarily for fine-grained communication between different cores. They are not used by the Flexicache system except as a source of extra scratchpad memory. The static-network router in each tile contains a 64 KB memory (the *Switch Memory* or *S-mem*) that can be accessed in exactly the same manner as the I-mem. Flexicache uses a small portion of the S-mem for storing some of its internal data structures.

One of the dynamic networks (the *Memory Dynamic Network* or *MDN*) is used to access external DRAM. To request a block of memory from DRAM, a core sends a small message containing the desired address. The appropriate DRAM bank then replies with a message containing the requested data. The request message may be initiated implicitly by a miss in the hardware data cache or can be explicitly created and sent by instructions executed on the core. When performing an explicit transaction, the core is free to continue executing other instructions while it waits for the response from DRAM. Flexicache uses explicit DRAM transactions to avoid interfering with the user program's use of the data cache.

The MDN is also used to access off-chip I/O devices. As with DRAM, messages containing commands or data are exchanged between a tile and an I/O device. One very important device in our prototype Raw systems is the *Host Interface*. Because Raw does not yet have a fully-featured operating system, it relies on a separate host computer to provide most standard input/output services, such as a file system and display of output. The Host Interface serves as the bridge between the software running on Raw and the host computer's operating system. When a program on Raw needs to perform a system call, it sends a message containing the call's arguments to the Host Interface. The Host Interface performs the actual system call on the host computer and returns any results in another MDN message. Because the MDN is used for both proxied system calls and DRAM access, the Flexicache system must take steps to ensure that the two uses do not conflict.

3.5 Chapter Summary

This chapter provided a brief introduction to the Raw microprocessor and described several key features that are relevant to the Flexicache system. These features include 1) the instruction memory architecture, 2) interrupt mechanisms, 3) external DRAM access mechanisms, and 4) method for proxying system calls to a host machine.

Chapter 4

Flexicache Algorithms and Implementation

This chapter describes our implementation of the Flexicache system for the Raw microprocessor. It provides a detailed description of the operation of the rewriter and runtime system. Although some of the implementation details presented here are specific to Raw, most are applicable to other processors as well. It is assumed that the reader is already familiar with the overview of Flexicache presented in Chapter 2.

4.1 User Program Modification (The Rewriter)

As mentioned in Section 2.2.1, the preprocessing portion of the Flexicache system is performed by a binary rewriter operating just before the final program link. The rewriter performs two primary tasks, breaking the user program up into cache blocks and redirecting control-flow paths that leave each block to the runtime system.

4.1.1 Cache Block Formation

Although basic blocks have several properties that make them a good choice for cache blocks, they also have the drawback that they can be composed of any number of instructions. They usually average between five and seven instructions (see [97] and Section 7.3.2) but can be much larger or smaller. For example, the Mediabench benchmark suite contains basic blocks ranging in size from 1 to 862 instructions. Having to manage variable-sized blocks would significantly increase the complexity of the runtime system. However, if the

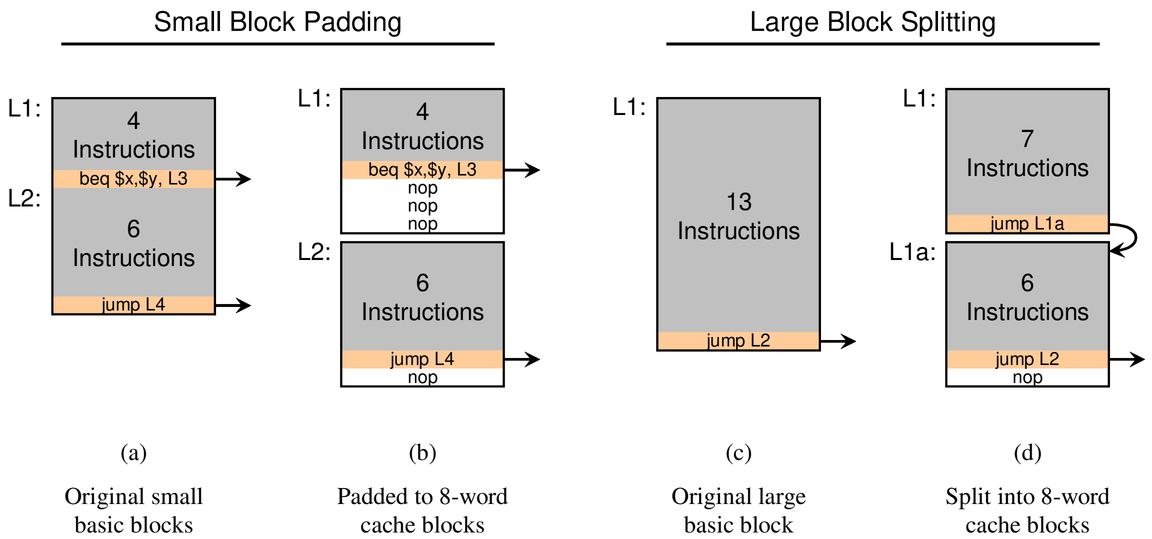


Figure 4-1: Examples of transformations done by the rewriter to fit variable-sized basic blocks into fixed-size cache blocks. The two basic blocks shown in (a) are padded with `nop`s to form the two cache blocks in (b). The single large basic block in (c) is split into two cache blocks in (d). A fall-through jump is inserted at each split to preserve correctness when these blocks are loaded separately.

system could use fixed-size cache blocks, then it would know *a priori* how many DRAM fetches are required for each block, how many blocks will fit in the cache, how much space to reserve for internal data structures, etc. Therefore, we compromise and place individual basic blocks within fixed-sized cache blocks. Basic blocks that are smaller than the fixed size are padded with `nop` instructions (although any instructions can be used since they will never be executed). Basic blocks that are larger than the fixed size are split into multiple cache blocks by inserting jumps. Figure 4-1 shows examples of these operations.

When designing a system with fixed-size cache blocks, there is a tension between making the blocks large or small. Using large cache blocks will increase the amount of space wasted on padding but will decrease the size of any data structures that are proportional to the number of blocks in the cache (see Section 4.2.1). Using smaller cache blocks will reduce padding for small basic blocks but increase the number of splits that must be performed for larger basic blocks. Since splitting basic blocks introduces additional costly calls to the runtime system (at the end of each cache block), the cache block size should allow most basic blocks to fit without splitting. Furthermore, if the cache block size is a power of two, there is the additional benefit that many of the calculations done by the runtime system can be implemented as bit masks or shifts (which are faster than general arithmetic operations on most processors). To balance padding, splitting, and data structure size requirements, cache blocks were initially fixed at 16 instructions. Based on results gathered from this initial system, the option to use eight-instruction cache blocks was added later.

As a final note, it was found that padding cache blocks with something other than `nop` instructions facilitates debugging of the software caching system. While debugging, it is common for the system to jump to an incorrect address and accidentally execute padding instructions. If those instructions are inert, the processor will sequence through them and continue on into other cache blocks. This produces very confusing behavior and usually allows the processor to continue running for a long time before the error is noticed, making it hard to find the source of the problem. By substituting an instruction that causes the processor to stop immediately or signal an error, the bug can be located much more quickly.

4.1.2 Control Flow Modification

The rewriter must ensure that a call is made to the runtime system whenever the flow of control exits a cache block. There are two different types of control-flow events that can cause execution to leave a cache block: implicit fall-throughs due to sequential execution and explicit control-flow instructions. Fall-through paths may be present in the original program basic blocks (following conditional branches or where a branch destination forced the start of a new block) or may have been created by the block splitting described above. With fall-through paths, it is necessary to insert new instructions to perform the call to the runtime system. In the case of control-flow instructions (CFIs), the existing instructions can usually be modified to perform the call. This is important since any extra instructions inserted will negatively impact performance at runtime.

The modifications required for fall-throughs and most CFIs are fairly simple. In general, there are three basic tasks: change the CFI destinations so that they point to the runtime system, convert CFIs to save their link address, and make sure each CFI uses an absolute addressing mode. The first task is self-explanatory. The goal of the second task is to provide a pointer for the runtime system back to the instruction that just made the call. The link address is the physical address of the instruction following the CFI and is usually used for procedure call returns. Here, it is used to locate the caller and perform runtime optimizations that will be discussed in Chapter 6. The final task is to make sure the CFIs use absolute addressing. This means that the destination value stored in the instruction is

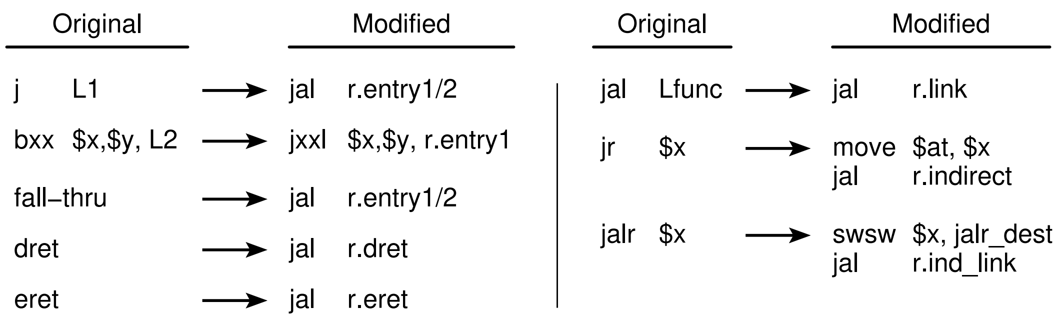


Figure 4-2: Modifications to control-flow instructions performed by the rewriter. “Fall-thru” is not an actual instruction but represents an implicit control-flow path from one block to the next. Labels starting with “r.” refer to runtime system entry points. r.entry1/2 means either r.entry1 or r.entry2.

a complete physical address, rather than an offset from the current program counter value. Since the runtime system remains resident in I-mem at a fixed location, a CFI using an absolute address (and the cache block containing it) can be placed anywhere in the I-mem and still jump correctly to the runtime code. This eliminates the need for costly patching of CFI destinations when blocks are loaded from DRAM.

On Raw, all three basic changes can be made in-place and without inserting additional instructions for all simple unconditional jumps (`j` instructions) and conditional branches (`bxx` instructions). As seen in Figure 4-2, unconditional jumps are simply replaced by jump-and-link (`jal`) instructions. On Raw, all opcodes that start with a “j” use absolute addressing. Thus it is clear that a `jal` has all the required attributes. Similarly, conditional branches are replaced by equivalent `jxxl` instructions. These instructions perform the same comparison as the original branch but specify their destination using an absolute address and also save the link address if the jump is taken. (While they are probably unique to Raw at this point, it would be easy to add them to most architectures since all of the required datapaths should exist for other purposes.) If these compound instructions were not available, the same result could be achieved using a combination of a traditional branch and a jump-and-link, albeit with a reduction in efficiency (see Section 8.3). Note that the fall-through path of the branch (and any other fall-through path) is handled by a separate call to the runtime system. First, an unconditional jump to the following block is inserted at the fall-through point. Then, the jump is processed like any other unconditional jump.

The careful observer will have noticed that the destination addresses for the modified CFIs in Figure 4-2 are not all the same. This is because the runtime system contains several different entry points. These entry points will be discussed in more detail in Section 4.2 but essentially, there are different entry points for different types of instructions. Each modified CFI must point to the correct entry point so that the runtime system will know how to handle each call. The instructions discussed above use the primary entry points, *entry1* and *entry2*. However, there are also special-purpose entry points for other instructions. Function calls (`jal` instructions) and interrupt returns (`dret` and `eret` instructions) are simply replaced with jump-and-links to the appropriate entry points. Indirect jumps (`jr` and `jalr` instructions) get their destinations from a register so this value must be passed to the runtime system. An instruction is inserted to set up this argument and then the CFI is replaced with a `jal` to the *indirect* or *ind_link* entry points.

Besides jumping to the runtime system, each modified CFI must somehow communicate its originally intended destination address. As mentioned above, an indirect jump passes its address through a register or other location at runtime. A similar approach could be taken with other CFIs, whereby instructions are inserted before the jump to the runtime system that load the destination address and pass it to the runtime system. However, that involves inserting additional instructions that bloat the user code and add overhead to CFIs. Instead, the rewriter builds a table (the *destinations table*) that contains the virtual destination addresses. At runtime, this table is stored in DRAM along with the rest of the program and can be accessed by the runtime system.

Each row of the table contains the destination addresses from a single cache block. Because each cache block contains one basic block, it can have up to two exits. Thus, the destinations table needs to have two columns. When a particular CFI is modified, if its destination address is stored in the first column, it jumps to the *entry1* entry point. If it is stored in the second column, it jumps to *entry2*. This tells the runtime system how to retrieve the correct value when it receives a call. In the case of function calls, it is assumed that the first column holds the procedure address and the second column holds the return address. Indirect jumps must pass their values at runtime instead because their destinations are not static.

Because the rewriter modifies CFIs to save their link address in the link register (LR), the value that the user program had in LR will be lost. Therefore, to maintain correctness, the rewriter inserts additional instructions to spill LR to a dedicated storage location in local memory. Whenever there is an instruction from the original program that writes a value to

LR, an instruction is inserted after it that writes that value to the spill location. Similarly, an instruction is inserted before any instruction that reads LR to retrieve the value from the spill location. Since LR is typically used only to hold the return address during procedure calls, it is very rarely directly manipulated and therefore, very little spill code is actually needed. Generally, only procedures that call other procedures (and therefore save LR on the stack) require spill code.

Finally, an additional note on addressing modes: Any CFIs that jump to a location within their own cache block should use a PC-relative addressing mode. Since these CFIs do not cause execution to leave a block, they do not need to be redirected to the runtime system. Using a PC-relative offset for internal branches will allow the cache block to execute correctly, without patching of destination addresses, no matter where it is placed. These CFIs can occur when a basic block jumps back to itself to form a tiny loop or in a hypothetical future system where cache blocks contain more than one basic block.

4.2 Runtime System

The runtime system is responsible for performing all of the cache management activities during program execution. These activities include 1) keeping track of the cache blocks that are currently present in the I-mem, 2) receiving requests from the user program for cache blocks and checking to see if they need to be loaded from DRAM, and 3) fetching new blocks from DRAM and deciding where to store them and now to make space for them if the I-mem is full. To accomplish these tasks, careful thought must be given to runtime data structures, the interface used for block requests, and replacement policies.

4.2.1 Data Structures

Choosing appropriate bookkeeping data structures is crucial to the efficient operation of the runtime system. These structures must provide very fast lookups to ensure that cache hits have minimal overhead. The size of these structures is also an important consideration because they should be stored in on-chip memory rather than DRAM to ensure that they can be accessed quickly. If stored in on-chip data memory, the bookkeeping structures will compete with the user program's data for space. Alternatively, a portion of the I-mem can be set aside for bookkeeping. Since the I-mem is already managed by the runtime system, this is simply a matter of reducing the amount of storage available for cache blocks. Either way, excessively large structures will hurt performance by robbing space from other uses.

In Flexicache, we adopt a variation of the second approach. To avoid polluting the data cache the runtime system data structures are stored in the switch instruction memory. The switch instruction memory can essentially be viewed as an extension of the I-mem since it is the same type of memory and is accessed in the same way (see Section A.1.3). Although the switch memory is quite large, the runtime system's data structures occupy only a small portion of it, roughly equivalent to the amount of extra space that would be required to replace the I-mem with a traditional hardware cache.

The primary function of the runtime system is to keep track of the blocks that are currently in the cache and transfer control to the appropriate block given a requested virtual address. This is accomplished using a hash table that maps a virtual address to the physical address where that block is currently loaded. Using a hash table provides fast lookups but has the potential problem of conflicts. Since checking multiple entries sequentially (or following a chain of pointers) would be very costly in software, conflicts are resolved by simply overwriting the old entry. This means that the block referenced by the old entry will still be in the cache but the runtime system will have forgotten about it. If the runtime receives a request for the block in the future, another copy will be loaded. Therefore, a hash table with a load factor[1] of about 0.5 is used to keep the probability of conflicts low. Increasing the size of the hash table any further would result in it being mostly empty and wasting a lot of space. Because the load factor is kept constant, the size of the hash table is proportional to the number of blocks that will fit in the cache. While it may initially seem foolish to allow multiple copies of a cache block to remain in the I-mem taking up space, the advantages of this scheme will become clearer after the discussion of replacement policies (Section 4.2.3) and the chaining optimization (Section 6.2).

In addition to the hash table, the runtime system uses another table (the *block data table*) to store information about each block that is currently in the cache. When a block is loaded into the cache, its data is entered into the row corresponding to the location where it is stored. As shown in Figure 4-3, there are four 32-bit entries in the block data table for each block. The first is the virtual address of the block and is needed to remove the entry

[1]The *load factor* of a hash table is a measure of how full it is expected to be. A hash table that can hold 100 entries but only contains 75 has a load factor of 0.75.

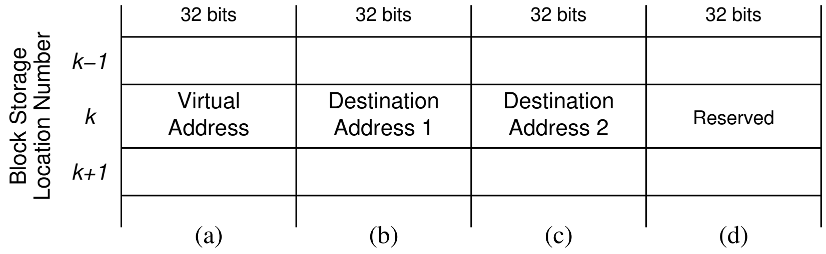


Figure 4-3: Block Data Table: Runtime data structure used to store information about each cache block that is currently present in the I-mem. When a block is loaded into storage location k, its meta-data is placed in row k of this table. This meta-data consists of the virtual address of the block (a) and a copy of the block's row of the destinations table (b,c).

from the hash table when a block is evicted. The second and third entries are copies of the destination addresses from the destinations table. These addresses are copied from the destinations table (in DRAM) into the block data table (in switch memory) when a block is loaded. This allows them to be accessed more rapidly since they will be needed during almost every call into the runtime system. The final entry is reserved for an optimization that will be discussed in Section 6.2.

Both the hash table and the block data table have a number of entries proportional to the number of cache blocks stored in the I-mem. Because the cache block size is fixed, the number of blocks that will fit in the I-mem is fixed. Thus, the size of all of the locally stored data is static and proportional to the size of the I-mem itself rather than the size of the user program. This means that accessing and updating this data is fast and efficient and that a large user program cannot cause these structures to overflow the local memory. Therefore, the total application size is limited only by the size of the DRAM.

4.2.2 Entry Points

When a modified CFI makes a call to the runtime system, it jumps to one of several entry points. As mentioned previously, the runtime system has different entry points to handle different types of instructions (Table 4.1). The primary function of the entry point routines is to determine what virtual address is being requested. This address will then be passed to the core of the runtime system. Multiple entry points are needed because different CFIs communicate their addresses in different ways. For example, some CFIs have their destination addresses stored in the destinations table while others pass their destinations directly in a register.

Name	Corresponding CFIs	Destination Address Source	Additional Actions
entry1	Branches (`bxx`), Jumps (`j`)	DT:1	
entry2	Branch (`bxx`) fall-throughs	DT:2	
link	Function calls (`jal`)	DT:1	Copy DT:2 to LR_spill
indirect	Function returns (`jr`)	Passed in $at	
ind_link	Indirect function calls (`jalr`)	Passed in S-Mem	Copy DT:2 to LR_spill

Table 4.1: Characteristics of the different runtime system entry points. The second column indicates which CFIs get converted to calls to each entry point. Branches become two calls to the runtime system, one for the *taken* path and one for the *fall-through* path. DT:1 and DT:2 refer to the first and second columns, respectively, of the destinations table. S-Mem refers to the switch memory and LR_spill refers to the LR spill location.

There are five primary entry points in the runtime system: *entry1*, *entry2*, *link*, *indirect* and *ind_link*. *Entry1* is used for branches and unconditional jumps that have their destination address stored in the first column of the destinations table. *Entry2* is the same but uses the second column. *Link* is for function calls (*i.e.*, `jal` or jump-and-link instructions) and also gets its destination from the first column of the destinations table. *Indirect* is for indirect jumps (*i.e.*, `jr` or jump-through-register instructions) and expects the destination to be passed through $at (the assembler temporary register). *Ind_link* is for indirect function calls (*i.e.*, `jalr` or jump-through-register-and-link instructions) and expects the destination to be passed through a dedicated location in switch memory. Besides this primary function, some entry points perform additional actions in order to replicate the behavior of the original CFI that was replaced by the rewriter. The *link* and *ind_link* entry points both retrieve the link address from the second column of the destinations table and store it in the LR spill location.

The entry points that retrieve values from the destinations table need to know which row of the table to read from. In reality, the entry points do not directly access the destinations table at all. When a cache block is loaded from DRAM, the miss handler routine in the runtime system also fetches the corresponding row from the destinations table. These data are stored in the block data table row that corresponds to the location where the cache block is placed in I-mem. It is this copy of the data that is actually accessed by the entry points. However, the entry point now needs to know which row of the block data table to access. This is where the runtime system makes use of the link address that was stored by the modified CFI. Because cache blocks are a fixed size, the entry point can easily calculate which block storage location (or *slot*) the modified CFI is in. This slot number can then be used to index into the table.

Once an entry point has found the correct destination virtual address, it passes it on to the routine that looks up this address in the hash table to determine if the block is already loaded. If there is a hit in the hash table, control is transferred to the corresponding physical address. If the destination address is not found in the table, the miss handler is invoked.

4.2.3 Miss Handler and Management Policies

The miss handler is responsible for fetching cache blocks from DRAM and implementing the cache management policies. When the miss handler is invoked, it first sends requests for the missing cache block to external memory. While it waits for the response, it selects a location for the new block and updates the hash and block data tables. If the I-mem is already full, an old block will need to be evicted to make space for the new one. When the response arrives, the new block is copied into the selected location in the I-mem. The miss handler then fetches the block's row from the destinations table[2] and stores the values in the block data table. The runtime system then jumps to the new block and the cycle repeats itself.

The selection of a location for the new block is one opportunity to improve on the methods of a hardware cache. In software it is feasible to implement a fully-associative cache rather than a typical set-associative cache. A fully-associative cache allows any cache block to be placed in any slot within the cache. This allows for much greater flexibility in terms of which blocks can be present in the cache simultaneously. A fully-associative cache may contain any set of blocks that will fit within the cache. This flexibility can greatly reduce the number of cache misses and virtually eliminate thrashing (repeatedly loading and evicting a set of blocks because they conflict with one another). In Flexicache, there is still a possibility that blocks will conflict in the hash table and be loaded multiple times. However, Section 7.3.1 will show that these conflicts can be minimized. Because there are no restrictions on where a block may be placed, the miss handler simply selects the next available slot if the cache is not full.

If the cache is full when a new block is fetched from DRAM, an old block must be evicted

[2]For convenience, Flexicache currently uses the data cache to retrieve these values, resulting in minor cache pollution (see Section 7.1.5). They could be fetched using explicit messages instead, if desired.

to make room. In Flexicache, we have implemented two different replacement policies: *FIFO* and *Flush*. The *FIFO* policy evicts the oldest block in the cache while the *Flush* policy clears the entire cache and starts fresh. These are the two most common replacement policies used by the various systems that employ software code caches [38].

Although conventional wisdom indicates that FIFO is not a good replacement policy, it avoids the complications of tracking fragmented free space that occur when using an LRU or random policy with a fully-associative cache. Also, because this is an instruction cache rather than a data cache, the access patterns tend to be more sequential making FIFO more appropriate. The FIFO policy has additional advantages over LRU and random when used with the chaining optimization that will be described in Section 6.2.

The Flush policy is commonly employed by dynamic binary translators (see Section 9.2 and [39]). It requires less bookkeeping than FIFO because blocks do not need to be individually evicted. The FIFO policy requires the virtual address of each block to be stored in the block data table so that the appropriate entry can be removed from the hash table when that block is evicted. The Flush policy simply clears the entire hash table. Therefore, the virtual address can be eliminated from the block data table. The disadvantage of the Flush policy is that a lot of very recently used code is evicted. This has a tendency to increase the miss rate compared to the FIFO policy. However, the advantages of the Flush policy are only truly realized when it is combined with the optimizations that will be described in Chapter 6.

4.3 Chapter Summary

This chapter delved deeper into the implementation of the Flexicache system. It explained how variable-sized basic blocks are transformed into fixed-size cache blocks through splitting and padding. It also described, in detail, how the rewriter redirects control-flow paths that leave those cache blocks to the runtime system.

Next, the internal workings of the runtime system were discussed. The runtime system uses a hash table and the block data table to keep track of the blocks that have already been loaded into the I-mem. It has several different entry points that are used for different types of instructions. These entry points form a key interface between the runtime system and the user program, even though they are invisible to the application programmer. When the cache becomes full, the runtime system can use either the FIFO or Flush replacement policies to make room for new blocks.

Chapter 5

User Interfaces and Transparency

The system described so far is completely transparent to the application programmer. The rewriter and runtime system work together to hide the details of the memory hierarchy and provide the illusion of a single large, fast memory. This system is adequate for most simple well-behaved programs. However, there are some situations in which it may not be possible or desirable to maintain complete transparency. These require the establishment of interfaces that define how the user interacts with the software caching system. This chapter describes several key features that we have identified that require external interfaces.

The features described in this chapter are not required for basic software instruction-caching systems running simple applications like those found in most benchmarks suites. As a result, many academic projects tend to overlook or ignore the difficulties associated with supporting them. It is only when software systems are used in real hardware environments or on larger, practical applications that the limitations of the system are found and addressed. In many cases, these "real life" issues are the most challenging and interesting to solve.

The interfaces and features presented here were all developed as a result of pushing the Flexicache system to handle new applications. The ability to pin code in the cache was first needed when we began to support C applications that perform file I/O. As explained below, it was used to avoid conflicts over resources between Flexicache and the I/O system calls. Support for cached interrupt handlers was added to aid the implementation of rMPI [73], an MPI-compliant message passing library for Raw. With interrupt support, the Flexicache system allows large, complex rMPI applications to be run on the actual Raw hardware. Support for self-modifying code is needed to allow rgdb (Raw's port of the gdb debugger) to set accurate breakpoints. Rgdb replaces the instruction at the breakpoint with jump to a pinned-down routine that executes the original instruction and traps into the rgdb runtime system. This behavior can be performed "magically" in the Raw simulator but debugging real applications on the actual chip requires rgdb and Flexicache.

5.1 Pinned Code

One reason that a programmer might choose to give up transparency is to specify pieces of code to be pinned in the cache. Pinned sections of code are permanently resident in the cache and cannot be evicted. There are several possible reasons for pinning code. The first is that pinned code will have consistent, predictable timing every time it is executed. Because the code is permanently resident, there is no chance of a cache miss creating an unexpected delay. This is an important feature for programs with real-time deadlines. Pieces of the program where timing is critical can be pinned down while others, such as initialization or user interface routines, can be cached normally in the remaining memory.

The second reason is to optimize program performance. Pinned code is always ready to be executed without delay and cannot conflict with any other cache blocks. Depending on the particular usage patterns of a program, it may improve overall performance to keep certain pieces of code resident. The cache is constantly guessing which cache blocks will be needed again soon. Since it has no information about the future, it guesses that the oldest blocks in the cache will not be needed. However, the programmer may have more detailed knowledge of his application's behavior and know that certain code will be needed repeatedly. In this case, pinning that code might lead to fewer overall cache misses. In addition, because the entire pinned section is always resident, there is no need for the rewriter to modify any of the internal structure or control-flow instructions. This eliminates all caching overhead and provides for the best possible performance in that section.

The final reason to pin code is more unusual and may not occur in all software caching systems. Pinning of code can be used to avoid conflicts for resources between the runtime system and the user code. For example, on Raw, the MDN network is used for both memory traffic and to send proxy system calls to the host computer (see Section A.2.3). If the user code that is sending a proxy system call is interrupted by a cache miss, the system call message and the DRAM request messages will interfere with each other and produce incorrect behavior. Thus, all of the code that sends and receives system call messages is pinned down so that it will not be interrupted.

Although it is possible to implement code pinning in hardware caches, our software cache is able to provide better support for it. With a hardware cache, there are typically only a small number of ways (between one and four) in each set. When lines are pinned in the cache, the ways occupied by those lines become unavailable for other lines and the associativity of the cache is effectively reduced. This increases the probability of thrashing significantly. In a fully-associative cache such as ours, pinning a block has a negligible effect on the placement of other blocks and thrashing is avoided. Furthermore, an n-way set associative cache permits only n-1 lines to be pinned in each set because at least one way must remain available for other lines. A fully associative cache, on the other hand, can pin as many blocks as will fit in the cache, without regard to address conflicts [37]. Flexicache goes one step further by treating pinned code separately from cached code. Space is statically allocated for the pinned code and the remaining space is managed as an instruction cache. Thus, the pinned code requires no additional runtime bookkeeping and does not occupy space in the runtime data structures. This increased code-pinning flexibility is a key advantage of our software caching system.

Flexicache allows the programmer to pin code on the granularity of procedures. In other words, functions must be treated as either pinned or cached in their entirety. This provides a clean, simple interface for both the programmer and the runtime system. A list of pinned functions is provided to the rewriter so that it can treat calls to those functions appropriately. The actual code for the pinned functions is not modified by the rewriter but is, instead, combined with the runtime library and the modified user code during the final program link. At runtime, a special entry point in the runtime system is used to transition back and forth between cached and pinned code. There is an additional restriction that pinned functions may only call other pinned functions. Therefore, if a pinned function needs to call a second function, the second function must also be added to the list of functions to pin. This restriction could be lifted but that would require the pinned code to be processed by the rewriter and additional runtime checks to be used at all function returns. Although we did not investigate it thoroughly, we felt that the restriction was preferable to the extra complexity and overhead.

The rewriter handles calls to pinned functions in much the same way as it handles calls

Name	Special Use	Destination Address Source	Additional Actions
pinned	Calls to pinned functions	DT:1	Perform call, Return to DT:2
sys_int	System interrupts	Int vector table	Virtualize EX_PC
user_int	User interrupts	Int vector table	Virtualize EX_UPC
eret	Sys int handler return (`eret`)	Virtual EX_PC	Re-enable system interrupts
dret	User int handler return (`dret`)	Virtual EX_UPC	Re-enable user interrupts

Table 5.1: Characteristics of the special-purpose runtime system entry points. For the two interrupt entry points, the destination (handler) address is stored in the interrupt vector table. The `eret` and `dret` routines make use of the virtual versions of EX_PC and EX_UPC created by `sys_int` and `user_int`. DT:1 and DT:2 refer to the first and second columns, respectively, of the destinations table.

to cached functions: the original `jal` instruction is modified to jump to the runtime system. However, if the rewriter sees that the destination of the function call CFI is on the list of pinned functions, it uses the *pinned* entry point (Table 5.1) rather than the *link* entry point. The address of the function is still placed in the first column of the destinations table and the address of the following cache block is still placed in the second column. However, in this case, the function's address is a physical address. Since the function is pinned, its address can be statically determined and no runtime translation is required.

The *pinned* entry point to the runtime system is somewhat more complex than the other entry points. Essentially, it forms a wrapper around the call to the actual pinned function. The initial steps of retrieving the destination and link addresses from the block data table are the same as in the *link* entry point. However, since the destination address is already a physical address, a hash-table lookup is not needed. Instead, the entry point code directly performs a function call to the pinned function. When the pinned function returns, the entry point must verify that the cache block it needs to return to is in the cache. The return address is simply the link address that it retrieved earlier from the block data table. This address is passed to the main portion of the runtime system for lookup in the hash table and execution continues just as it does with the other entry points.

5.2 Interrupt support

Because an interrupt is essentially a jump to a new piece of code, and because Flexicache is responsible for managing a program's code, some amount of interaction with the caching system is required to use interrupts. Interrupt handler code may be either pinned down or cached. In the case of a pinned handler, the only required interaction with the I-caching system is to add the handler to the collection of pinned routines. The user program can then set up the interrupt vector table, configure interrupts and enable/disable them as it normally would.

Using a cached handler requires a great deal more support from the rewriter and runtime system but very little effort on the part of the programmer. To use a cached handler, the interrupt vector table must be configured to invoke the runtime system rather than simply jump to the handler code. This requires some knowledge of the internal workings of Flexicache and would be difficult for a programmer to do himself. Therefore, the runtime system provides a routine that the user program can call to setup the appropriate entry in the vector table. Once an interrupt handler is invoked, it runs and makes calls to the runtime system just like any other piece of code. This means that the rewriter needs to modify it along with the rest of the program. It also means that the rewriter will need to include support for any special "interrupt return" control-flow instructions. However, this will all be transparent to the programmer.

From a user perspective, using a cached interrupt handler is easy: The user program simply calls a special runtime routine with the interrupt number and the virtual address of the handler. This routine places three instructions in the interrupt vector table that (when the interrupt fires) pass the address of the handler to a special entry point (either *sys_int* or *user_int*, shown in Table 5.1) in the runtime system. The runtime system handles this request as it would any other: checking the hash table and loading the block if needed.

However, the entry point must perform an additional task first: The interrupt return address (saved by the hardware in EX_PC or EX_UPC when the interrupt fired) must be converted from a physical address to a virtual address. This is necessary because the execution of the handler could cause the code that was interrupted to be evicted from the cache. In this case, the virtual address will be required to reload the interrupted block from external memory. First, the entry point checks to see if the interrupt fired while the processor was executing pinned code or runtime system code. If the return address is located in permanently resident code, it does not need to be virtualized. If the return address is in cached code, it is used to retrieve the virtual address of the interrupted block from the block data table. This address and the offset of the return address into the block are stored in runtime system variables until the interrupt handler returns.

Interrupt handlers on Raw end with either a `dret` or `eret` instruction (depending on the interrupt type) to jump to the interrupt return address and re-enable interrupts. The rewriter translates these instructions into calls to additional runtime entry points. These entry points check the virtual return address calculated earlier and reload the interrupted block if needed. If the block needs to be reloaded, it will no longer be in the same place in the I-mem and the hardware interrupt return address will need to be corrected to correspond to the new location. Finally, the runtime system executes the appropriate `dret` or `eret` instruction to return from the interrupt and re-enable interrupts atomically.

In addition to adding the new entry points, the rest of the runtime system must also be carefully modified to work with cached interrupt handlers. All runtime system code must be protected from interrupts because it is not reentrant. If an interrupt were to fire during a runtime system routine, the additional calls to the runtime system required to load and execute the handler would corrupt the original state. The runtime system is not reentrant because it cannot assume that a stack has been set up. Therefore, all runtime variables are stored in hard-coded locations in switch memory. Interrupts must also be disabled to avoid having an MDN transaction interrupted by additional transactions to fetch the handler code. As a result, all entry points must save the interrupt state and disable interrupts. When transferring control back to user code, the runtime system must restore the previous interrupt state and perform the jump atomically. These changes add between seven and nine cycles to every call into the runtime system. All data in this thesis was collected with interrupt support included. However, for situations were interrupt support is not required, removing it would eliminate approximately 644 bytes from the runtime system (leaving more space for cache block storage) and improve the runtime system performance.

5.3 Self-Modifying Code

Self-modifying code provides challenges for many software and hardware systems alike; Flexicache is no exception. Ironically, Flexicache itself could be considered self-modifying code because it is constantly changing what is stored in the I-mem. Fortunately, self-modifying code is fairly rare in typical user applications. Since most programmers who use it are highly-skilled and aware that the practice may be hazardous, maintaining perfect transparency is not particularly important. A programmer who is willing to expend the effort to write self-modifying code will not be troubled by having to use a special interface to do so.

Because pinned code is not modified or managed by the Flexicache system, it can be modified freely at runtime. Programs requiring extensive use of self-modifying code can pin the routines that need to be modified. For cached code that needs to be modified, the runtime system provides a special routine that can be called by the user program to write an instruction. It simply takes an address and the value to write. There are some limitations on the types of modifications that can be made which will be discussed below. Writes to the code segment could be discovered and modified automatically by the rewriter but, in some ways, it seems better to make the programmer use a manual interface so that he realizes that he is doing something out of the ordinary. In this case, the rewriter should detect writes to the code segment that do not use the manual interface and signal an error.

The standard challenge with self-modifying code is making sure that all copies of the code within the memory hierarchy are consistent. First, the special runtime routine writes the new value into DRAM to ensure that all future fetches of the code retrieve the correct value. Then, it must ensure that no stale code can be executed from the cache. With the basic system described so far, this can be done easily by looking up the address of the modified cache block in the hash table and clearing the entry if it is found. This will cause a new copy of the block to be loaded the next time it is needed. However, this will not be sufficient once the optimizations from Chapter 6 are added. The runtime system will need to ensure that any copies of the cache block that are in the cache have the new value.

There are three options to accomplish this: evict all copies of the block from the cache, flush the cache completely, or update all copies of the block in-place. Using the optimizations, it will not be possible to evict an arbitrary, individual block. In fact, even without the optimizations, the FIFO and Flush replacement policies do not normally permit removal of arbitrary blocks. Thus, the first option is not possible. The second option works perfectly well but is very expensive, both because of the time it takes to clear the entire hash table and because the entire cache will need to be reloaded. However, this method is simple and may be preferred if code modifications are infrequent. The final option can be implemented by searching the virtual address column of the block data table for occurrences of the modified block. The search will be very slow but the overall performance may be better because no blocks will need to be reloaded. Flexicache currently implements the second option because our only self-modifying applications are not concerned with performance.

Note that, even with the support we have described, there are limits to the modifications that are allowed. First, one must be very careful when generating the address where a modification is to be made. Because the rewriter inserts instructions into the program in many places, the relative positioning of many instructions will have changed. The safest way to generate an address is to place a label at the desired location. The rewriter will move the label along with the instruction, ensuring the correct address. Unless the programmer is very familiar with the workings of the rewriter and is willing to double-check the final binary, he should not try to specify locations using an offset from a label. Second, the programmer should not try to modify control-flow instructions because they must be statically analyzed and modified by the rewriter. Attempting to change them at runtime will most likely result in disaster. Applications that need to dynamically modify control-flow should pin the routines that need to be modified unless the programmer is an expert on the internal workings of Flexicache.

5.4 Uninterruptible Regions

In the existing Flexicache system, pinning code is the only way to prevent instruction cache misses and avoid conflicts for MDN resources. However, some of the results of pinning code could be achieved using a lighter-weight alternative: In some situations it may be sufficient to guarantee that no instruction cache misses occur in the middle of a piece of code. In other words, the code may need to be fetched from DRAM initially but, once it starts executing, it will run without interruption. This feature could be used to prevent conflicts for resources between user code and the runtime system and might even be adequate for some real-time applications. For example, with uninterruptible regions, the proxy system call routines would no longer need to be pinned, thereby freeing up additional space for cache blocks.

Providing this functionality would require two features. First, the programmer must be able to designate the regions of his program that are uninterruptible. Second, the runtime system needs to be able to fetch and store all of the code for each region as a unit. This might mean prefetching all of the cache blocks that make up an uninterruptible region or it might mean fetching all of the code as a single large cache block. In addition, if uninterruptible regions and cached interrupt handlers are used in the same application, interrupts must be turned off during the uninterruptible code.

Although we have not fully implemented this feature, most of the groundwork has been laid. Marking code and disabling interrupts can be accomplished simultaneously on Raw. The Raw tool chain supports two instructions called `mlk` and `munlk`. The `mlk` instruction signals the beginning of an uninterruptible region and disables interrupts. The `munlk` instruction signals the end and turns interrupts back on. When executed on the actual Raw chip, these instructions just turn interrupts on and off. However, the special opcodes are used to communicate the uninterruptible nature of the code to the rewriter.

Loading of the uninterruptible region as a unit can be accomplished using the macroblock feature described in Chapter 6. Actually, even without macroblocks, the system would be able to handle uninterruptible regions that were small enough to fit in a single cache block. Since this is not likely to be large enough for most uses, the macroblock feature allows multiple cache blocks to be combined into a single large block. For uninterruptible regions that are larger than the maximum size supported by macroblocks, it would still be necessary to fall back on pinning the routine.

5.5 Chapter Summary

This chapter discussed several key interfaces between the user program and the Flexicache system. These interfaces arise from situations where it is not possible or not desirable to maintain perfect transparency of the software instruction-caching system. These types of interfaces are usually not needed for simple programs but become crucial when dealing with larger programs in "real life" environments.

Pinned code is kept resident in the cache at all times to enhance performance or avoid conflicts for resources. *Interrupt support* gives the user access to all of the interrupts on Raw and allows handlers to be either pinned down or cached. Support for *self-modifying code* can allow a skilled user to dynamically optimize or modify his program behavior. Finally, *uninterruptible regions* provide some of the benefits of pinned code without the need to permanently occupy I-mem space.

Chapter 6

Optimizations

The software instruction-caching system described in the preceeding chapters provides all the necessary functionality to run a wide variety of programs. However, the performance of this baseline system is generally very poor. This chapter starts by analyzing the baseline system performance and then describes several optimizations that have been added to Flexicache to improve this performance. These optimizations are

1) basic chaining,
2) function-call decomposition,
3) indirect-jump chaining,
4) macroblock fusion, and
5) LR-spill code rescheduling.

Along with the description of each optimization, results are presented showing the impact of that optimization on application performance.

6.1 Baseline System Performance

As shown in Figure 6-1, programs executed using the baseline Flexicache system typically take between 3.5× (for mcf) and 10× (for ammp) as long to complete as they would using a similarly-sized hardware instruction cache. This graph plots the number of cycles needed to complete the application using the Flexicache system, divided by the number of cycles needed by a 32 KB, 2-way set-associative hardware instruction cache. This normalization is performed to compensate for the vastly different execution times of the different benchmarks and to provide a performance comparison with a traditional hardware-cache architecture.

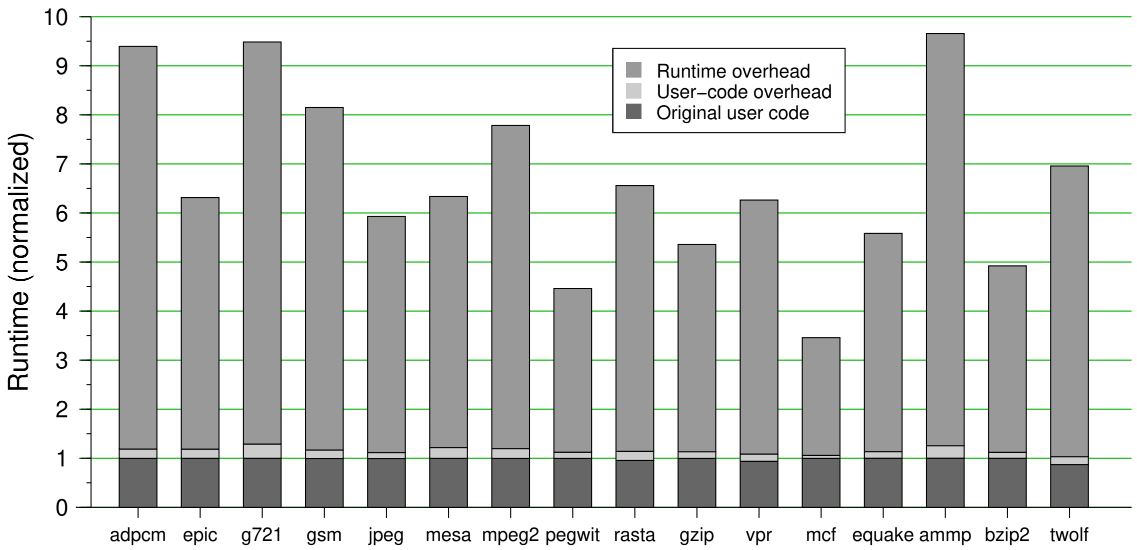


Figure 6-1: Performance of unoptimized baseline system. Run times are relative to hypothetical hardware I-cache performance. See Section 7.1.1 for experimental methodology.

Additional detail on the experimental methodology used to collect this data can be found in Section 7.1.1.

Figure 6-1 also shows the breakdown of the total runtime into three different components: original user code, runtime overhead and user-code overhead. The "original user code" component represents the time needed to execute the application's original instructions, assuming that they are all immediately available to the procesor pipeline and do not need to be fetched from DRAM. This is an absolute lower bound on the total application run time. Any cycles beyond this lower bound are considered "overhead" because they represent additional time needed to perform instruction caching operations. The overhead in these applications comes from two different sources: calls to the runtime system and instructions inserted into the user program code by the rewriter. The first factor accounts for the vast majority of the overhead in the baseline system and is the primary focus of the optimizations given here. As the runtime-system overhead is reduced, however, the user-code overhead becomes more significant and requires optimizations of its own.

6.2 Chaining

One of the disadvantages of placing only a single basic block in each cache block is that it results in very frequent calls to the runtime system. Since basic blocks are fairly short,

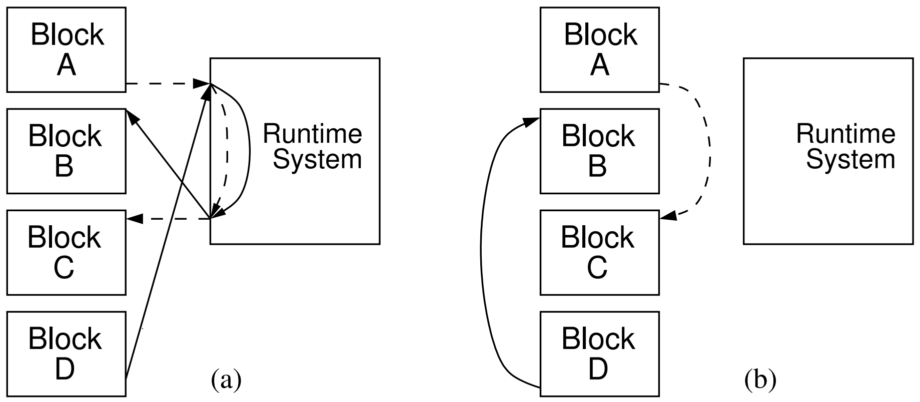


Figure 6-2: Example of jumps between blocks (a) before chaining and (b) after chaining. In (b), the costly calls to the runtime system are skipped.

a call to the runtime system is generated for every five to seven instructions of the user's code, on average. Because even a hit in the runtime system takes between 40 and 45 cycles, the overhead is substantial and performance is poor.

Chaining is an established technique in the dynamic binary translator community [19, 97, 24, 7] that cuts out unnecessary calls to the runtime system by modifying the code in the cache. Here, we apply it to the new domain of software instruction caching. When the runtime system loads a block into the cache, it goes back and changes the destination of the modified CFI (control-flow instruction) that caused that block to be loaded so that it jumps directly to the new block. The next time that CFI is executed, it will skip the call to the runtime system, jump directly to the next block, and incur no overhead (Figure 6-2). The runtime system can find the CFI it needs to modify using the link address that the modified CFI saves (see Section 4.1.2). Chaining can be performed both when a new block is loaded and when a block is requested that is already present in the cache. In fact, the runtime system can create a chain every time it is called, except when the original CFI was indirect (*i.e.*, the target address was stored in a register) [19].

The difficulty with chaining is that it complicates deallocation. When a block to which a chain has been created is evicted from the cache, every CFI that points to it must be changed back to a call to the runtime system. This allows the block to be reloaded in case it is needed again. In order to perform this *unchaining*, the runtime system must keep track of the CFIs that have been modified to point to each block. When a CFI is modified, a back-pointer to the CFI and the previous destination address are stored with the target

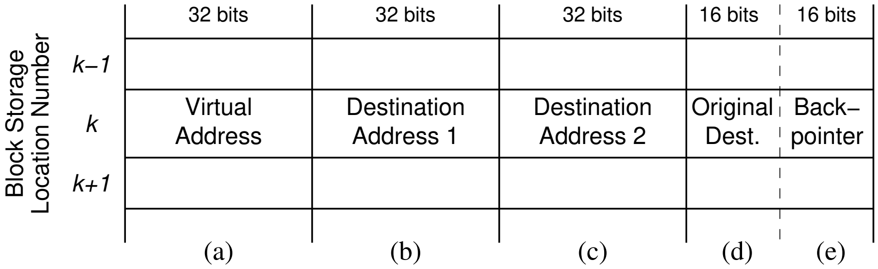


Figure 6-3: Block data table updated with new unchaining information. Since both the original destination and backpointer are 16 bit values, they can be packed into a single table entry.

block's data (see Figure 6-3). When the target block is evicted, the back-pointer is used to find the CFI that was modified and the stored destination field is used to change it back to a call to the appropriate runtime entry point.

In general, there can be any number of chains to a particular block thereby requiring a variable amount of storage for the unchaining information. However, to reduce the size and complexity of the data structures, storage is statically allocated for only one chain per block (as suggested in [20]). Since the back-pointer and previous destination address are both less than 16 bits, they are packed together and stored in the fourth column of the block data table. When this space is full, either no new chains can be created to this block or the old chain must be undone to make room for the new chain. However, when using a FIFO replacement strategy, it is not necessary to keep track of chains that go from older blocks to newer blocks (see Figure 6-4(b)). If a chain goes from an older block to a newer block, it is guaranteed that the block containing the jump that was modified will be evicted from the cache before the block that is the destination of the jump. Therefore, it will never need to be unchained. Thus, with the single chain storage slot, we can create an unlimited number of chains from older blocks and a single chain from a newer block for each block in the cache.

Now the beauty of the Flush replacement policy becomes clear. If the entire cache is cleared at once, all chains which have been created are automatically thrown out. Thus, it is not necessary to keep track of or undo any chains. Now there are no limits on which jumps can be chained and each chain takes less time to create. In our implementation it takes six cycles to create a chain with either FIFO or Flush. With FIFO, however, it takes up to 24 additional cycles to determine whether a chain needs to be recorded and record it if

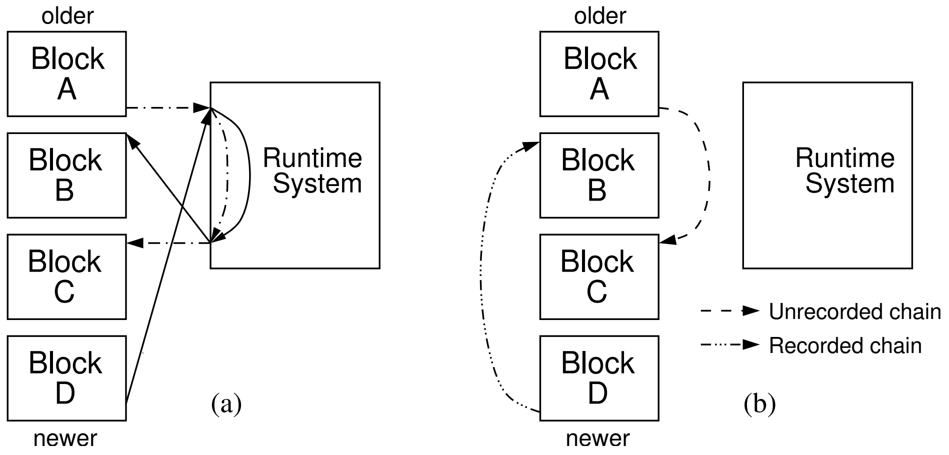


Figure 6-4: Example of jumps between blocks (a) before chaining and (b) after chaining. When using FIFO replacement, the chain from Block A to Block C does not need to be recorded because A will be evicted before C. When using Flush replacement, no chains are recorded.

it does. Also, because it is no longer necessary to track chains, the unchaining information can be eliminated from the block data table, freeing up more space to store cache blocks.

The disadvantage of eliminating unchaining information is that it prevents arbitrary cache blocks from being evicted individually. If an arbitrary block were evicted, dangling chains could be left in the cache. When a new block is loaded into the freshly-emptied space, the dangling chains would jump to the wrong piece of code. By taking advantage of the properties of the replacement policy, the runtime system is prevented from performing any evictions outside that policy. This is the primary reason why the old block is not evicted when there is a conflict in the hash table (as discussed in Section 4.2.1). However, the use of chaining also means that the old block is not useless. Even though the runtime system no longer knows about it, any chains to the block will still be able to use it. This will reduce the number of future requests for the block and therefore, the probability that another copy of the block will need to be loaded. In practice, we find that aggressive use of chaining can dramatically reduce the number of hash table conflicts and resultant misses (see Section 7.3.1). Thus, even though the simple hash table essentially makes the cache direct-mapped, with chaining the overall behavior of the cache is much closer to fully-associative.

Figure 6-5 shows an example of the dramatic improvements that chaining provides. The first bar in each cluster is the baseline data from Figure 6-1. The second bar shows the

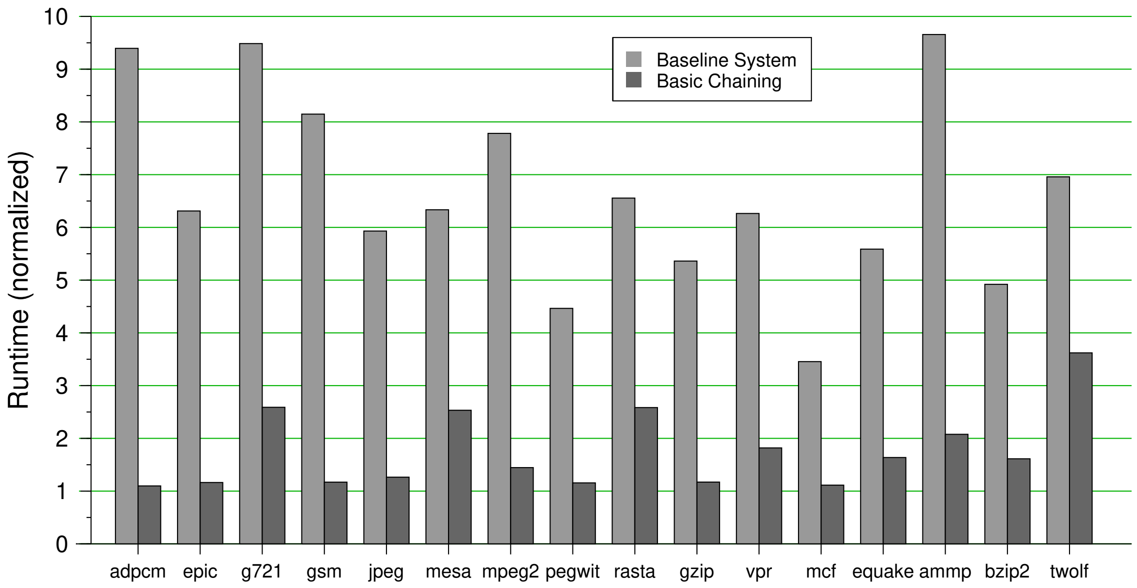


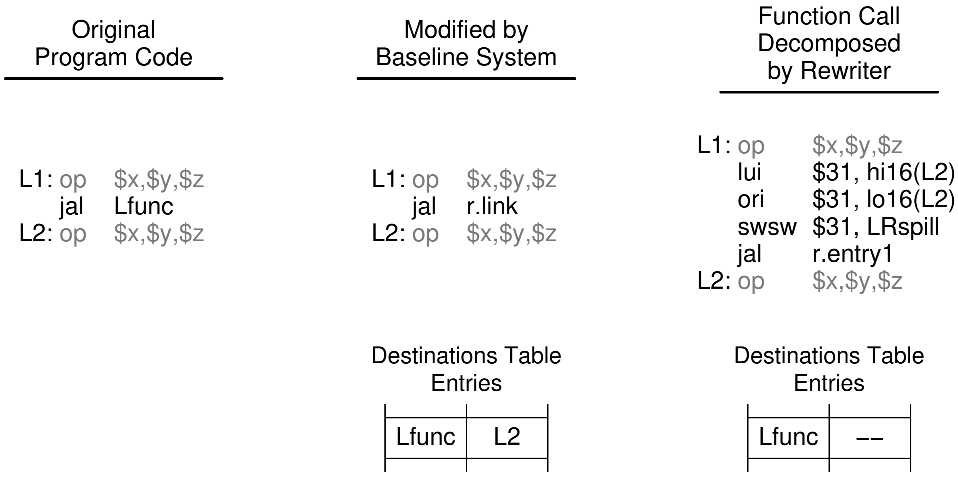
Figure 6-5: Performance improvement using basic chaining with FIFO replacement. The first bar in each cluster is the unoptimized baseline system. The second bar shows the result of adding the basic chaining optimization.

improved performance when using the chaining optimization. In this example, chains are created on both hits and misses but only for CFIs that jump to the *entry1* or *entry2* entry points in the runtime system. This includes CFIs that were originally conditional branches or plain unconditional jumps but not function calls or returns. Here, the FIFO replacement policy is used and new chains are aborted if the chaining information slot in the block data table is already full.

6.3 Function-Call Decomposition

Function calls on Raw (and many other processors) are performed with a special control-flow instruction called a *jump-and-link*. This instruction actually performs two operations: jumping to the function's address and saving the return address[1] (also known as the *link* address) in a register. Under software instruction caching, both of these addresses must be virtual addresses. The function's virtual address must be processed by the runtime system just like the destination address of any jump instruction. The return address stored in the link register must be a virtual address because the runtime system must be able to make sure that the necessary cache block is loaded when the function returns. If the physical address

[1]The return address is simply the address of the instruction following the function call instruction.



(a) (b) (c)

Figure 6-6: Examples of rewritten function calls in (b) the baseline system and (c) when decomposing function calls. In (b), the task of saving the return address (L2) is performed by the runtime system using the value in the second column of the destinations table. r.link and r.entry1 refer to runtime system entry points.

of the instruction following the function call was stored instead and the block containing the call was evicted during the course of the function's execution, then the function would return to the wrong code. However, there is no single processor instruction that can both make the call to the runtime system *and* store the virtual return address. Therefore, the software instruction-caching system must perform these two tasks using multiple instructions.

The baseline system attempts to minimize the introduction of new instructions into the user code by handling both of the function call tasks in the runtime system. Both the function address and return address are stored in the destinations table and retrieved by the special-purpose *link* entry point (see Figure 6-6(b)). The problem with this approach is that it makes chaining function calls much more complicated. If the runtime system took the normal approach to chaining these calls (changing the destination field of the instruction that called the runtime system), then the jump to the function code would be performed but the virtual return address would no longer be saved. Therefore, extra instructions would need to be inserted into the cached copy of the block to perform the save. At the very least, this would take longer than creating a normal chain. However, in some cases, it would not be possible to insert the extra instructions due to insufficient empty space in the block. Either the runtime system would have to leave some calls unchained or the rewriter would need to split the basic block and add empty space to guarantee that the chain would succeed. As a result of these difficulties, the runtime system does not attempt to chain any calls to the *link* entry point.

There is a simple alternative, however, to this complicated mess. The rewriter can insert the instructions that save the return address itself as shown in Figure 6-6(c). Because the rewriter would have needed to reserve space for those instructions anyway, no additional space is wasted. Because the return address must be saved by separate instructions whether the call has been chained or not, no additional work is being done. Some of the work has simply been moved from the runtime system code into the program code. Now that the return address is saved separately, the rewriter can treat the jump-and-link as if it were just a jump. In essence, the original compound jump-and-link instruction has been decomposed into its separate parts: a save of the return address followed by a regular jump. The regular jump can now use the same entry points as the other jumps and be chained in exactly the same way. The *link* entry point can be completely removed from the runtime system, simplifying it and freeing space for cache block storage. The extra complexity of handling the compound instruction is now in the off-line rewriter rather than the performance-critical runtime system.

Currently, Flexicache only decomposes jump-and-link instructions but the technique is applicable to other instructions as well. For example, on Raw, it could also be used to break jump-through-register-and-link instructions into a save of the link address followed by an indirect jump. (This has not been done already because these instructions are fairly rare in most applications so the incremental gain would be small.) On other systems, it might be useful for loop instructions that decrement a counter and perform a branch. Compound instructions exist in processors because the hardware is able to perform both operations in a single cycle. However, using software instruction caching, it takes multiple instructions to perform these same tasks, wherever they are done. If the tasks are sufficiently independent, there is no reason to continue to treat them as a single, monolithic operation. Instead, it may make sense to decompose the compound instructions into their constituent operations early on in the workflow. This allows the two tasks to be independently optimized and the complexity of the system to be reduced by reducing the number of fundamental CFI types.

The true benefit of decomposing function calls (from a performance standpoint) is that it allows them to be chained like the other CFIs. Our initial instinct was that function calls might not be frequent enough to benefit from chaining. However, the results shown in

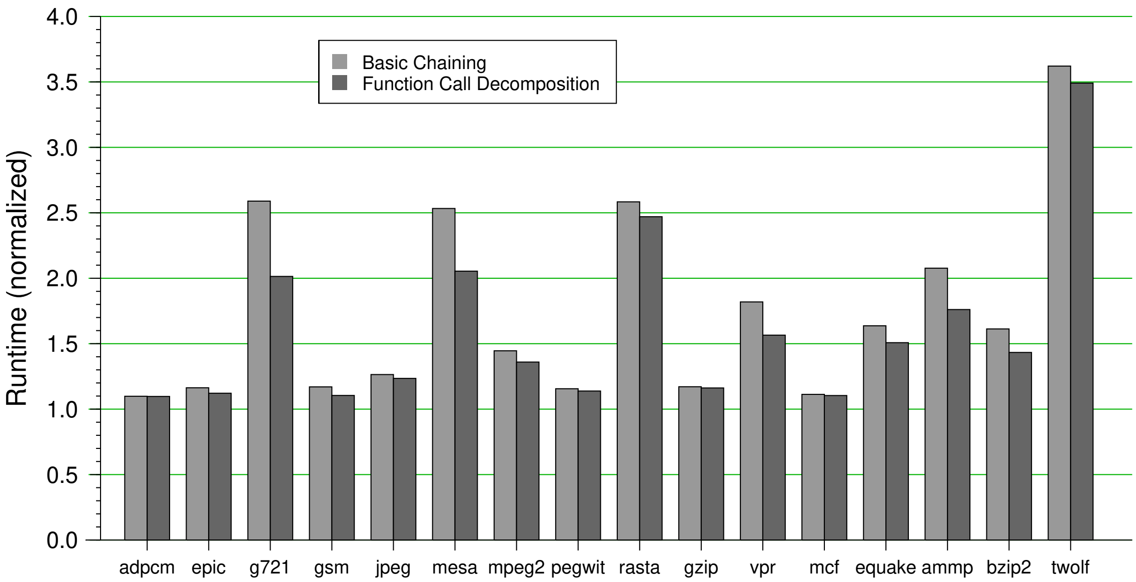


Figure 6-7: Performance improvement using the function-call-decomposition optimization. The first bar in each cluster uses basic chaining with the FIFO replacement policy. The second bar shows the result of adding the function-call-decomposition optimization.

Figure 6-7 clearly show otherwise. Those benchmarks showing the greatest improvement (g721, mesa, vpr and ammp) tend to perform function calls very frequently. However, every benchmark benefited to some degree, even those that perform relatively few function calls.

6.4 Indirect-Jump Chaining

Although indirect jumps cannot be directly chained, it is still possible to use chaining to optimize them. The problem with indirect jumps is that they might go to a different address each time they are executed while a chain goes to a single, fixed address. However, since indirect jumps are usually used for function returns and most functions are called from only a few places in the program, each indirect jump will likely go to a small number of different addresses. By separating out these addresses, they can each be chained individually. We use the same basic technique as DAISY [27] to accomplish this. The indirect jump is replaced with a sequence of instructions that compares the jump address to various individual addresses and executes a normal jump if it finds a match (see Figure 6-8). These normal jumps can then be chained as with any other jump. In our current implementation, this sequence is built up dynamically at runtime (see Section 10.1 for other possibilities). The rewriter remains unchanged and continues to replace indirect jumps with calls to the

Before Indirect Jump Chaining

```
L1: op    $x,$y,$z
    swlw  $at, LRspill
    jal   r.indirect
    nop
    nop
    nop
    nop
    nop
    nop
```

(a)

After Indirect Jump Chaining

```
L1: op    $x,$y,$z
    swlw  $at, LRspill
    lui   $31, hi16(L2)             } Indirect chain
    ori   $31, lo16(L2)             } to block L2
    jeql  $at,$31, L2-physical      }
    lui   $31, hi16(L3)             } Indirect chain
    ori   $31, lo16(L3)             } to block L3
    jeql  $at,$31, L3-physical      }
    jal   r.indirect
```

(b)

Figure 6-8: Example of indirect-jump chains created by the runtime system. Here, chains have been inserted for two virtual addresses: L2 and L3. It takes two instructions to load a full 32-bit virtual address. L2-physical represents the physical address where block L2 is currently stored. In this case, the link address stored by the `jeql` instruction is not used.

indirect entry point of the runtime system. When the runtime system receives an *indirect* call, a check for the requested address and a chain to the target block is inserted in the calling block. The original call to the runtime system is left at the end of the sequence of checks to handle any new addresses in the future.

In essence, the address is being prescreened to see if it matches a block that we have already seen. The drawback to this approach is that the prescreening takes time and space. If an indirect jump goes to many different addresses, it can take longer to do all of the individual comparisons than it would have to just call into the runtime system and perform the hash table lookup. As a compromise, we adopt the heuristic that the sequence may only grow to fill any remaining space in the fixed-size cache block. However, some blocks may have little or no empty space, preventing any chains from being created. To address this problem, the rewriter can be given a minimum amount of empty space for blocks containing indirect jumps. This ensures that some minimum number of individual comparisons can be performed. If a block's empty space is below this threshold, additional space can be created in two different ways. First, if the macroblock feature described below is enabled, additional empty blocks can be added after the indirect jump to meet the minimum. If this is insufficient or macroblocks are disabled, the rewriter can split the basic block and move the indirect jump to the beginning of a new block. In our experience, threshold values between zero and three comparisons provide good performance on most benchmarks. A more detailed analysis is given below.

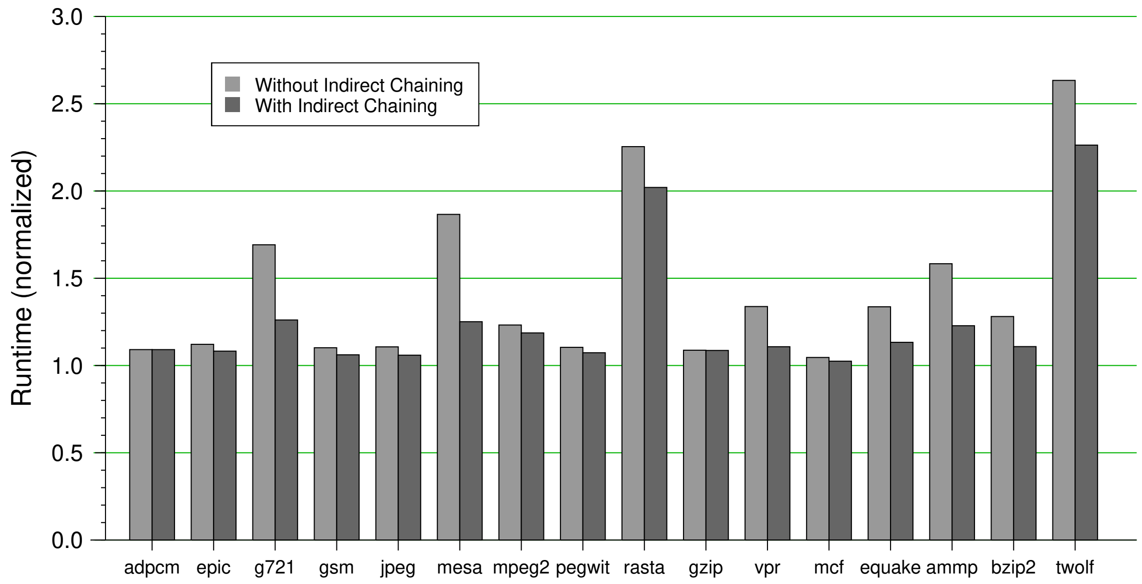


Figure 6-9: Performance improvement using indirect-jump chaining optimization. The first bar in each cluster uses basic chaining, function call decomposition and the Flush replacement policy. The second bar shows the results of adding the indirect-jump chaining optimization with an empty-space threshold of three comparisons.

Given the impact of chaining function calls shown above, it should come as no surprise that chaining function returns can also improve performance significantly. In fact, the improvements shown in Figure 6-9 closely mirror the improvements seen in Figure 6-7. The results in Figure 6-9 were collected using an empty-space threshold of nine instructions. Since each address comparison takes three instructions, this provides enough space for three individual address checks. However, the optimal threshold level is dependent on the application. Figure 6-10 shows the effects of varying the threshold from zero to twelve instructions (zero to four address comparisons). A threshold of zero means that the rewriter simply leaves whatever space is naturally left at the end of a block and never moves the indirect jump to a new block. On some applications, changing the threshold can affect the overhead by as much as 5%. Increasing the threshold can allow more indirect jumps to be handled by chains rather than the runtime system. However, if many jumps wind up falling through to the runtime system anyway, then performance will be hurt by the extra comparisons that were done first. In addition, increasing the threshold will cause more blocks to be split or padded with extra space. This increases the total size of the program, increasing the space pressure within the cache and requiring additional fetches from DRAM.

Although it is possible to chain indirect jumps when using either the FIFO or Flush

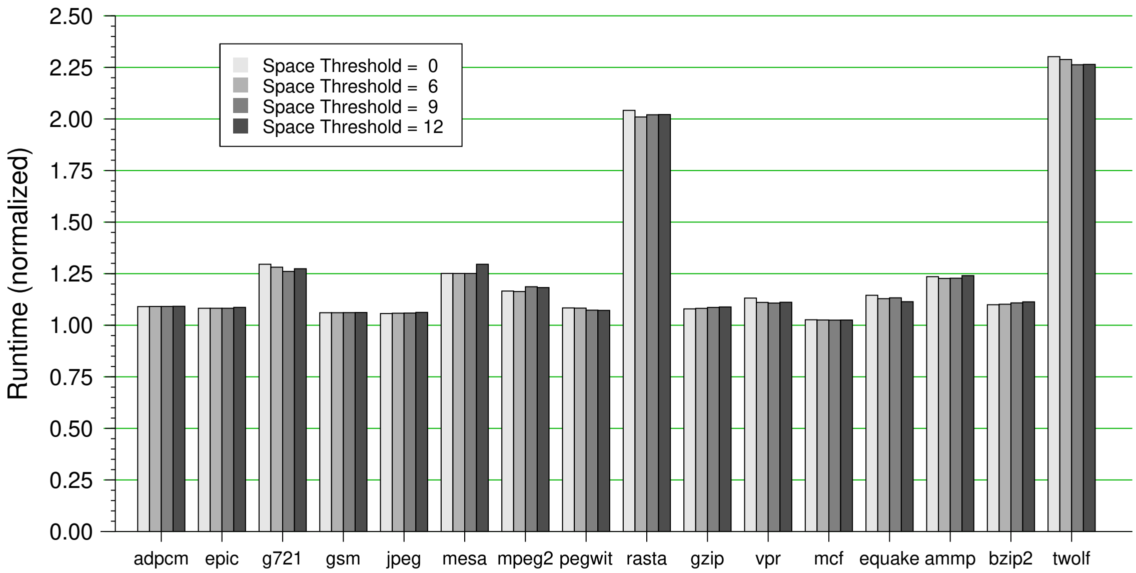


Figure 6-10: Effect of varying the empty space threshold for indirect-jump optimization. The first bar in each cluster shows the case where no minimum is enforced. Bars 2, 3 and 4 correspond to thresholds of 6, 9 and 12 instructions. Each address comparison takes three instructions.

replacement policies, this optimization has only been implemented with the Flush policy. It was not implemented with FIFO due to the additional complication and bookkeeping that would be required to undo the individual chains in the sequence. The fact that the Flush policy does not require unchaining allows the use of more aggressive optimization techniques that might not be practical if they needed to be reversed.

6.5 Macroblocks

One drawback of using fixed-size cache blocks is that large basic blocks must be broken up into multiple blocks that are each smaller than the limit. In the baseline system, the rewriter accomplishes this by inserting an unconditional jump at each point where the block needs to be split. The destination of the jump is simply the instruction after the split (*i.e.*, the next instruction). Because the jump is a control-flow instruction, it forces the end of one basic block and the start of a new one.[2] These jumps then get modified into calls to the runtime system just like pre-existing ones. This technique introduces overhead in two

[2]According to some definitions of the term "basic block," this jump would not actually force the end of a block. Since the jump is unconditional and there are no other ways to get to its destination, the instructions before and after it would all be considered a single basic block. However, for our purposes, the rewriter ends a basic block at every control-flow instruction.

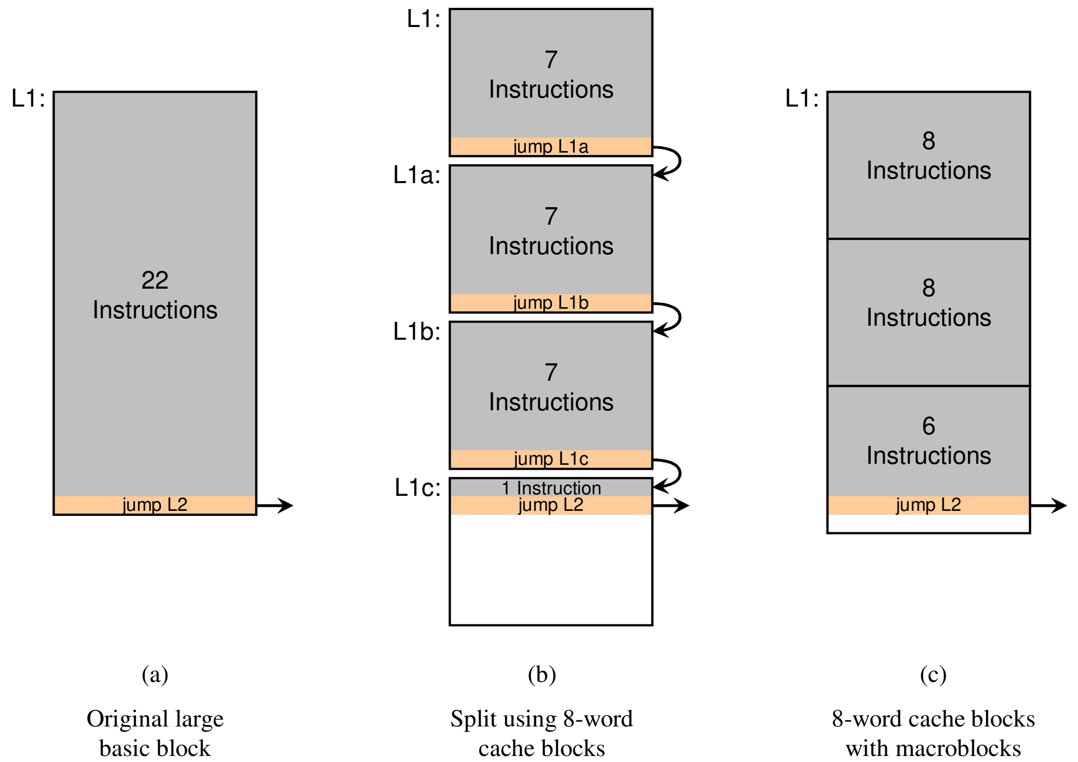


Figure 6-11: Examples of how the rewriter handles large basic blocks (b) in the baseline system and (c) using macroblocks. In (b), separate blocks are created and jumps are inserted between them. In this case, the additional jumps have forced the use of four cache blocks. In (c), three blocks are fused together to form a macroblock and no additional jumps are required. Note: The rewriter will later translate all jumps into calls to the runtime system.

ways. First, it creates multiple calls to the runtime system to fetch what is, logically, one block of instructions. The same sequence of cache blocks will always be loaded, stored and evicted together so the intermediate checks should not be needed. Each cache block also requires its own entry in the hash table, increasing the likelihood of hash table conflicts. In addition, when the blocks are not yet in the cache, each call will result in a separate cache miss with non-overlapping memory accesses. Second, the inserted jump instruction itself takes an extra cycle to execute. Even after the call has been chained and the runtime system overhead is eliminated, the extra jump instructions will continue to waste cycles. These overheads get worse as the cache block size is decreased (*e.g.*, from sixteen to eight instructions). However, simply increasing the size of all blocks is not a good solution because it will cause more space to be wasted in padding of small blocks.

The way to reduce or eliminate this overhead is to provide a mechanism that allows larger cache blocks to be used only for large basic blocks. This can be accomplished (while still maintaining the advantages of a fixed-size cache block) by fusing multiple small blocks together into a *macroblock*. Whenever the first sub-block of the macroblock is loaded, the runtime will automatically load the rest of the sub-blocks as well. As long as the sub-blocks are placed contiguously (and in their correct order) in I-mem, they will behave as a single large block and will not require jumps to be inserted between them. In effect, this gives Flexicache the ability to use variable-sized cache blocks, as long as the sizes are a multiple of the base cache block size.

Using this technique, the entire macroblock can be loaded with a single call to the runtime system. All of the administrative overhead of a runtime system call (disabling interrupts, saving and restoring the interrupt state, saving and restoring temporary registers, etc.) is incurred only once. In addition, the runtime system only needs to perform a single lookup in each of the block data and hash tables. When the macroblock is loaded from DRAM, the runtime system can send requests for all of the sub-blocks up front, thereby partially overlapping the round-trip delays to memory. Finally, since the sub-blocks are placed against each other in I-mem, execution can proceed from one sub-block directly into the next one, eliminating the need for any jumps between them.

Because macroblocks are built by combining multiple regular blocks, they are fairly easy to integrate into Flexicache. The rewriter can simply leave large basic blocks whole and pad all blocks until they are an integer multiple of the normal cache block size. It then calculates, for each macroblock, the number of cache blocks that are required to span it. Since all blocks use at least one cache block, only the number of additional cache blocks beyond the first one is actually required. This number (called the *autoload* value) is the key piece of information that needs to be communicated to the runtime system so that it can fetch the correct number of cache blocks from DRAM. One way to do this is to store the autoload value as meta-data for the first sub-block (possibly in a new column in the destinations table). When a cache miss causes the first sub-block to be loaded, its meta-data would also be loaded and could be examined to determine how many additional blocks to load. However, this requires the runtime system to wait for the meta-data to be returned from DRAM before it can request the additional blocks.

Ideally, the runtime system would know how many blocks to fetch as soon as a miss occurs. To accomplish this, the autoload value is embedded into the address of the first sub-block (as described below). Anywhere the address of the sub-block appears in the destinations table, it is replaced with a modified address that also contains the autoload value. Now, when the runtime system is presented with an address to jump to, it is also given the number of cache blocks to load if that address misses. The rewriter must be careful to find and modify all occurrences of the address in the user's program, including places where it has been stored in the data segment as a code pointer.

There are a couple of potential ways to embed the autoload value in a block's address. In many RISC processors (including Raw), all instructions are 32 bits long and are required to be word-aligned. Therefore, the two least significant bits of every code address are not used and make an excellent place to store the autoload value. The one difficulty with this choice is that there are only two bits available. However, we have decided to accept this restriction and limit macroblocks to no more than four regular cache blocks (requiring autoload values of up to three). Basic blocks that are larger than this limit are split into multiple macroblocks using the conventional jump insertion technique. Although this imposes some additional overhead, the total overhead for a large basic block will still be reduced by at least 4× versus a system without macroblocks. If this limitation was not acceptable (or the two low bits were not available), an alternative would be to use the most significant bits of the address word. These bits are also likely to be unused since programs rarely contain enough code to fill their entire addressable memory range.

One additional issue that must be considered is the destinations table. There must be a row in the destinations table for every cache block, even if that block is part of a larger macroblock. This is necessary to allow the runtime system to find the correct destinations given a cache block's address. However, since a macroblock is still just a single basic block, it only has one row's worth of data. It is important to make sure that the destinations data is present in the row corresponding to the sub-block that actually contains the calls to the runtime system. This will normally be the last sub-block but it is possible for it to be an earlier sub-block if empty blocks are deliberately added. It is also possible for a call to *entry1* to be in one sub-block and a call to *entry2* to be in the next sub-block. To simplify things, the destinations data for the macroblock is just duplicated for all of its sub-blocks. Now the runtime system will be able to find the proper data no matter which sub-block contains the actual call. In the future, the extra rows could be used to allow for macroblocks that contain multiple basic blocks and have more than two exit points.

Adding macroblocks can have a substantial impact on performance, particularly when

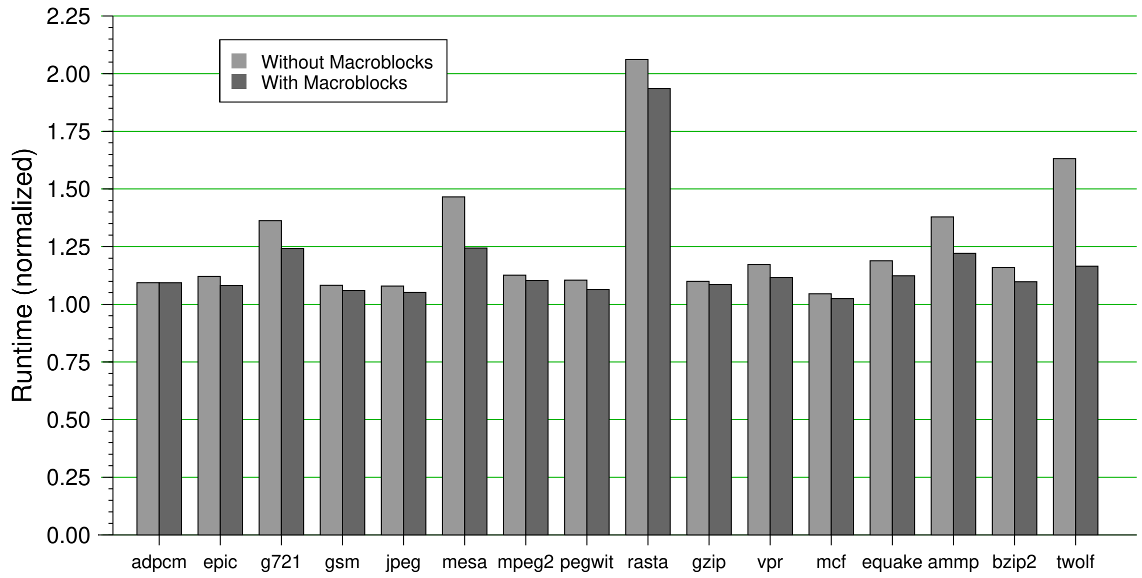


Figure 6-12: Performance improvement using macroblocks with 8-word cache blocks. The first bar in each cluster uses basic chaining, function call decomposition and indirect-jump chaining. The second bar shows the results of adding macroblocks, including the ability to increase the empty-space threshold.

they are used with a small cache block size. Figure 6-12 shows the result of adding macroblocks to a system using eight-word cache blocks. The initial system already employed all of the optimizations previously discussed. However, because of the small cache block size, indirect-jump chaining was limited to two address checks. Enabling macroblocks cut down the overhead for large basic blocks and allowed space to be reserved for additional address checks. In this case, the empty-space threshold was increased to allow three checks but it could have been even higher if desired.

6.6 LR-Spill Code Rescheduling

The optimizations described above have primarily focused on eliminating runtime-system calls to reduce the performance overhead of the Flexicache system. However, extra instructions that are inserted into the user code by the rewriter also contribute to this overhead. One of the largest sources of user-code overhead is the instructions inserted to spill LR described in Section 4.1.2. Recall that, in the baseline system, a spill load is inserted immediately before an instruction that uses LR and a spill store is inserted immediately after an instruction that writes it. In compiler generated code, the only instructions that use LR are storing it to the stack at the beginning of procedures. The only instructions that write LR are retrieving it from the stack just before the procedures return.

Although relatively few instructions are actually inserted, they can have a large impact on performance by causing additional pipeline stalls. The instruction used for a spill load has a latency of five cycles before the loaded value can be used by another instruction. Because it is inserted immediately before such an instruction, the processor will always stall on the load for the full five cycles. The spill store case is a little more subtle. Although the spill store itself executes in a single cycle, it introduces a use of LR where there may not have been one previously. This can cause the processor to stall on the instruction that writes LR if that instruction has a latency greater than one. Because the load instruction that is used to retrieve LR from the stack has a latency of three cycles, each spill store introduces three cycles of delay. Thus, every procedure that saves and restores LR takes an additional eight cycles to run after spill code insertion.

Some of this overhead can be removed by scheduling the spill instructions differently. Moving a spill load from immediately before a use to the beginning of its cache block allows other instructions to execute during the load latency period. If there are multiple spill loads in a block, only the first one needs to be moved; the additional loads can be eliminated. Conceptually, the value of LR is being retrieved from the spill location once for use by the entire block rather than individually for each instruction that needs it. However, if the use of LR is too close to the beginning of the block, this will not be sufficient to eliminate all stall cycles. In this case, the rewriter has a limited ability to reschedule the user code to reduce the stall. It attempts to move the instruction that uses LR down past other instructions as long as it can guarantee that no dependences are violated. Because the instruction that uses LR is usually a store to the stack, the rewriter must be fairly conservative when attempting to move it past loads and other stores. However, it will move the instruction past other stores that it identifies as stack saves.

The rewriter does not currently attempt to eliminate stalls associated with spill stores because of the smaller potential gain. However, the concept would be similar: spill stores should be moved to the end of the block (just before the call to the runtime system) and instructions that write LR might need to be moved earlier in the block to avoid all stalls.

This optimization generally produces more modest performance improvements than the others. However, after applying all of the previous optimizations, most of the runtime sys-

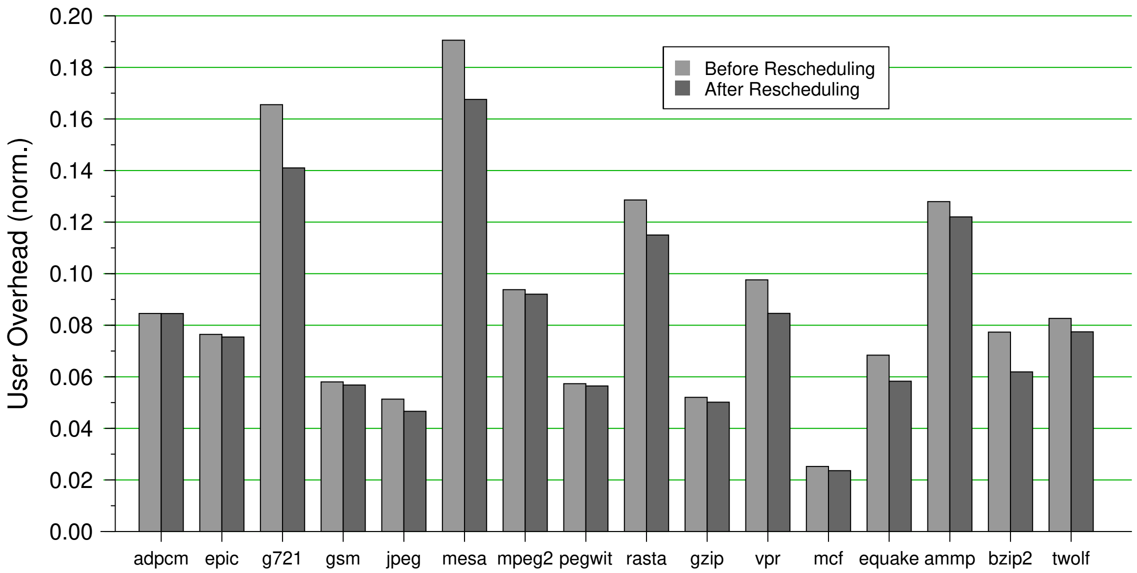


Figure 6-13: Impact of LR-spill code rescheduling on user-code overhead. Because this optimization has no effect on calls to the runtime system, this graph shows only the overhead introduced into the user code (the lightest bar segments in Figure 6-1). The vertical axis is the same scale as the previous graphs but this graph shows only overhead, not total runtime.

tem overhead will have been eliminated and the only remaining potential gains are small. Figure 6-13 shows the results of enabling spill code rescheduling in a version of the system using basic chaining, function call decomposition and the Flush replacement policy. The applications that show the most improvement perform large numbers of nested procedure calls that require LR to be saved on the stack. There are no situations where this optimization could hurt performance. Because this optimization eliminates overhead from the user code while the others target the runtime system, they generally do not interfere with each other and the improvements are additive. The exception is the macroblock optimization which allows the rewriter to reschedule instructions across cache block boundaries. This additional freedom increases the effectiveness of spill code rescheduling by allowing the rewriter to eliminate additional pipeline stalls.

6.7 Chapter Summary

This chapter presented five optimizations to the basic software instruction-caching system presented in Chapters 2 and 4. *Chaining* removes unnecessary calls to the runtime system by jumping directly to the next cache block when that block is guaranteed to be present in the cache. *Function call decomposition* breaks compound function-call instructions into multiple simple instructions. This accelerates the process of saving the return address and allows the jump to the new function to be chained easily. *Indirect-jump chaining* prescreens the jump target address to allow some indirect jumps to be chained. This is an important optimization for programs that perform many function calls. *Macroblocks* fuse multiple small cache blocks into a single entity. This reduces some of the overheads associated with using small, fixed cache blocks and increases the effectiveness of the indirect-jump chaining optimization. *LR spill code rescheduling* optimizes the insertion of required spill instructions to eliminate unnecessary processor stalls. Taken together, these optimizations are extremely effective at eliminating runtime overhead and result in performance comparable to a hardware cache on most benchmarks.

Chapter 7

Experimental Evaluation

This chapter presents additional experimental results for the Flexicache system implemented on the Raw microprocessor. Since the goal of every cache is to increase performance, performance is the primary focus of our evaluations. The impact of several optimizations and design choices are examined on a variety of benchmarks. However, power is also a major concern in many systems, particularly embedded systems that are more likely to lack hardware caches. Therefore, the energy consumption of Flexicache is evaluated and compared to a hardware cache. Finally, two additional system characteristics are examined: the number of conflicts that occur in the hash table and the amount of padding inserted to make all cache blocks a fixed size.

7.1 Performance

Providing good performance is an important goal for any caching system. In systems with explicitly-managed memories, peak performance is obtained through extensive program analysis, manual program partitioning and careful hand-coding. Programmers may be willing to sacrifice a small amount of performance for programming convenience but will prefer hand-optimization if the penalty is too great. This is especially true in the embedded domain where explicitly-managed memories are common. Performance is also related to energy consumption. The additional instructions that a software instruction cache must execute to manage itself will require additional energy. Therefore, reducing the number of instructions executed improves performance *and* decreases energy consumption.

Chapter 6 presented results with each optimization to demonstrate its effectiveness and show which applications benefit most from it. This section presents performance comparisons for some additional system features and pulls everything together to show the overall system performance. First, in Section 7.1.1, the experimental methodology used to collect all the results in thesis is described in detail. Second, in Section 7.1.2, we examine the performance implications of the two different replacement policies used in Flexicache. Third, in Section 7.1.3, we compare the performance of a system using 16-word cache blocks to one using 8-word cache blocks. Finally, in Section 7.1.4, we collect together all of the previous results to examine overall trends and find the optimal combination of optimizations and system parameters for each benchmark application.

7.1.1 Methodology

Flexicache has been used extensively on both the Raw cycle-accurate simulator (called "BTL" and pronounced "*beetle*") and the prototype Raw hardware systems. Although the speed of the actual hardware is ideal for testing and running long applications, the exact timing is highly variable. This is primarily due to the way that system calls are proxied to the host computer (see Section A.2.3). However, there is additional variability in the asynchronous boundary between the Raw chip and memory. Therefore, for maximum reproducibility and accuracy, all of the measurements of application run time in this section were collected using BTL. BTL has been extensively validated against the actual microprocessor and models it precisely on a cycle-by-cycle basis [87].

BTL also provides a deterministic and configurable model of the system in which the microprocessor is placed. For these experiments, a system that is functionally equivalent to the single-chip prototype system (see Section A.2) was used. However, there are two important timing differences. First, the round-trip latency to DRAM is 26 cycles rather than 60. Second, the proxied portion of each system call is executed instantaneously. The client portion is still executed on Raw and sends messages to the Host Interface to perform the actual call. However, those messages are interpreted and the required call is executed in a single simulation cycle as soon as the message leaves the Raw chip. As a result of this idealized model, I/O operations are essentially removed from the timing of the program. Therefore, the results collected focus only on the actual application code.

In addition to the timing differences, the simulator also has the advantage of a much richer set of profiling tools. This allows for the inspection and measurement of every as-

Benchmark	Operation	Command Line
adpcm	encode	rawcaudio clinton.pcm out.adpcm
epic	encode	epic test_image.pgm -b 25
g721	encode	encode -4 -l -f clinton.pcm
gsm	decode	untoast -fpl clinton.pcm.run.gsm
jpeg	decode	djpeg -dct int -gif -outfile testout.gif testorig.jpg
mesa	render	mipmap mipmap.ppm
mpeg2	encode	mpeg2encode options.par out.m2v
pegwit	encrypt	pegwit -e my.pub pgptest.plain pegwit.enc encryption_junk
rasta	extract	rasta -z -A -J -S 8000 -n 12 -f map_weights.dat -i ex5_c1.wav -o ex5.asc

Table 7.1: List of the Mediabench applications and command lines used in this study. Note that jpeg uses the -gif output option rather than the standard -ppm option.

pect of a program without influencing the results. For example, the simulator can track indirect-jump chains (see Section 6.4) and count the number of cycles spent executing them without having to insert additional code into the running program. However, for some cases where the goal was only to count the number of times an event happened (*e.g.*, hash-table conflicts), it was actually easier to insert a tiny snippet of code into the runtime system and rerun all the applications on the hardware. Thus, the results presented contain data collected from both BTL and the single-chip prototype system.

The simulator also allows us to simulate alternative processor designs for comparison. Besides the actual Raw design, BTL is able to simulate Raw with two alternative instruction memory models: a general-purpose hardware I-cache and a larger SRAM instruction memory. The hardware I-cache modeled is a 2-way set-associative cache with a 32 byte (8 word) line size and a 32 KB capacity. The larger SRAM model functions in exactly the same way as the actual Raw chip but has an I-mem capacity of 256 KB. Using the larger I-mem, the simulator is able to run all of the benchmarks from this study (except mesa) without requiring any form of caching. This provides a lower bound on the execution time since the application fits entirely in the fast local memory and no time is spent fetching code from DRAM.

As mentioned in Section 4.2.1, Flexicache stores some of its data structures in switch instruction memory instead of the main processor instruction memory. To create a fair comparison between the hardware cache and the software cache, this extra storage is limited to 40% of the I-mem size (or about 13 KB). This corresponds to the amount of area used for tags and control in the hardware cache (as seen in Figure 1-2). Thus the hardware and software cache models require approximately the same amount of chip area.

Benchmark	Integer/FP	Dataset	Description
164.gzip	CINT2000	lgred	Compression
175.vpr	CINT2000	mdred	FPGA placement and routing
181.mcf	CINT2000	smred	Single-depot vehicle scheduling
183.equake	CFP2000	lgred	Earthquake simulation
188.ammp	CFP2000	smred	Computational chemistry
256.bzip2	CINT2000	lgred	Compression
300.twolf	CINT2000	smred	Placement and routing

Table 7.2: List of the SPEC® CPU2000 benchmarks and datasets used in this study. All datasets are from the MinneSPEC workloads.

To evaluate the Flexicache system, two different benchmark suites were used: Mediabench [56] and SPEC® CPU2000 [21]. Mediabench is a set of benchmarks that provide a sampling of communications and media applications that are important for the embedded domain. Each benchmark typically includes separate applications for "encode" and "decode" operations as well as various command-line options. Table 7.1 shows which programs were run and any options supplied to that program. SPEC® CPU2000 contains a variety of integer and floating-point benchmarks that are primarily used to evaluate workstation and server-class processors. Unfortunately, only a small subset of SPEC® benchmarks are supported by Raw's research-grade toolchain. For example, Raw's toolchain does not support C++ or Fortran programs, or C's "long long" integers. All of the benchmarks that could be compiled and run easily were included. Since only a subset of the benchmarks are used, they should be considered a sampling of possible applications rather than a comprehensive range of applications. Another problem with SPEC® benchmarks is the size of the input datasets. These benchmarks are designed to run for significant amounts of time on fast modern processors. It is not practical to run such large benchmarks under simulation. Therefore, the MinneSPEC [53] alternative workloads were used. Table 7.2 lists the benchmarks and input datasets that were run.

7.1.2 Replacement Policy: FIFO vs. Flush

The first version of Flexicache used only the FIFO replacement policy. Initially it seemed to be a suitable alternative to LRU and worked reasonably well. However, after chaining and function call decomposition were added to the system, it became clear that it was hindering optimization. The culprit is not actually the FIFO policy itself but rather the additional bookkeeping required to evict blocks individually. Recall from Section 6.2 that bookkeeping

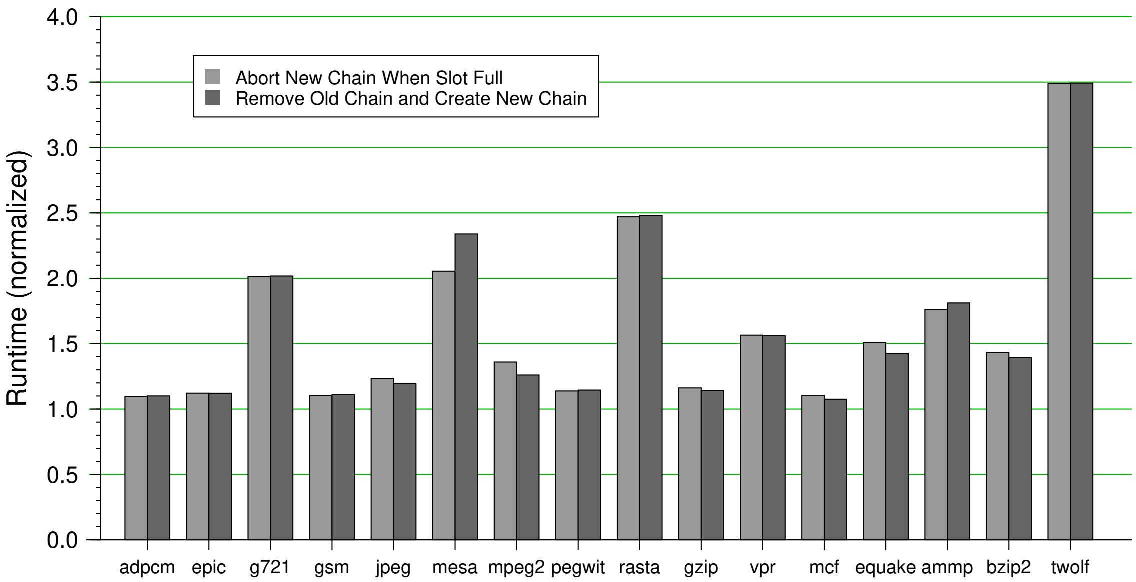


Figure 7-1: Comparison of two different strategies for managing the single unchaining information slot. The first bar in each cluster shows the result of aborting new chains after the slot is full. The second bar shows the case where the old chain is removed and replaced with the new one.

data must be stored for each chain so that it can be unchained if the destination block is evicted. To limit the costs of storing and accessing this data, only a single chain can be recorded for each block. Even though using the FIFO policy (instead of LRU or Random) allows some chains to go unrecorded, this single storage slot has proved insufficient. Some chains have to be aborted because the slot is already full. Even a small number of aborted chains can allow a very large number of calls to the runtime system to remain if they are located in frequently executed code.

To address this problem, an alternate method of dealing with a full storage slot was implemented. Rather than abort the new chain, the runtime system undoes the old chain (thus freeing up the storage slot) and then creates the new chain, thereby replacing the old chain with the new one. The results of making this change are shown in Figure 7-1. In half of the benchmarks, performance remained essentially the same. In benchmarks where the new chain replaced an old chain that was no longer needed (or at least would not be needed for a long time), performance improved. However, when two control-flow paths were used alternately, their chains would constantly replace each other, incurring additional overhead and preventing either chain from actually being used. This resulted in a decrease in performance. In all cases, there were still many calls to the runtime system that were

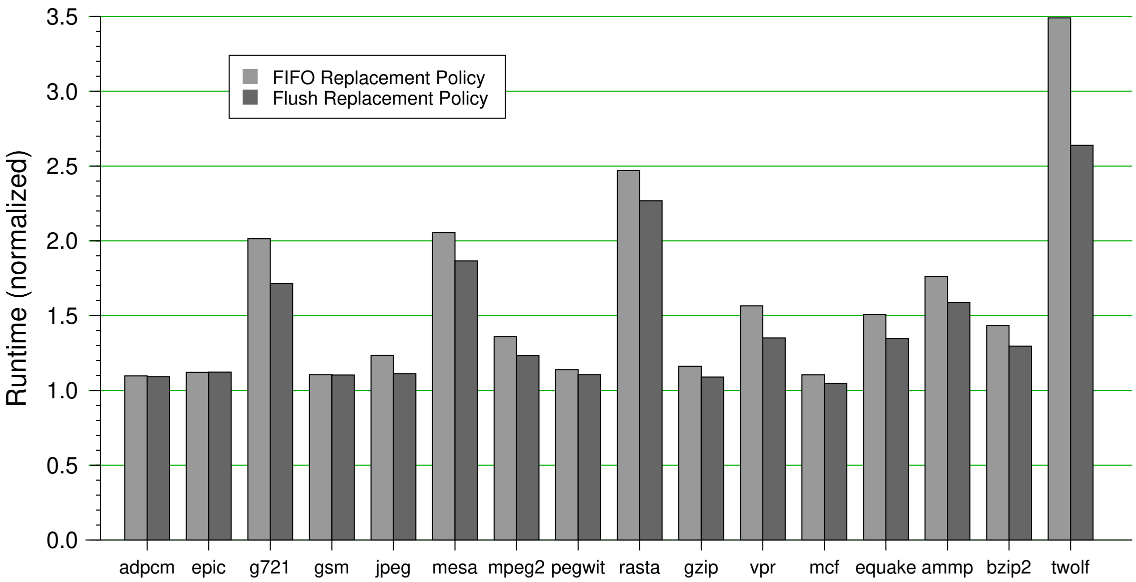


Figure 7-2: Comparison of FIFO and Flush replacement policies. The first bar in each cluster uses the FIFO policy while the second uses the Flush policy. Both bars perform chaining on hits and misses and use function call decomposition.

not optimized away.

Based on these results, it was clear that a better solution was required. One option that was considered was increasing the number of storage slots for unchaining information. However, this was rejected because the runtime bookkeeping data structures were already consuming a large fraction of the local storage space. Allocating additional slots would take space away from cache block storage. In addition, increasing the number of slots would only partially solve the problem. There could always be some cache blocks for which the number of slots was insufficient. A scheme allowing a variable number of storage slots would prevent this but would require more complex data structures (such as linked lists) and sophisticated local memory management. These data structures would have additional space overhead and much longer access times than the single, statically allocated slot.

Therefore, rather than attempt to patch the existing policy's limitations, an entirely new replacement policy (the Flush policy) was implemented. The Flush policy is discussed in Section 4.2.3 and Section 6.2 but the important point here is that it does not require any chains to be recorded. This allows an unlimited number of chains to be created and makes it faster to create those chains. It also allows the space previously allocated to store unchaining data to be used to cache blocks. On the other hand, the Flush policy increases the miss rate because it evicts many blocks that have only been in the cache a short while. Figure 7-2 shows the difference in performance between using the FIFO and Flush policies. In this example, the version of the system using Flush did not reclaim the storage space used for unchaining information. Therefore, any benefits are derived entirely from the additional chains that can be created and the reduced overhead for creating those chains. A substantial improvement is seen in most benchmarks. All of the benchmarks benefited primarily from the additional chaining but rasta, vpr, ammp and twolf also benefited significantly from the reduced cost of creating chains. Epic, adpcm and gsm showed little improvement because the increased miss rate offset the gains from improved chaining.

7.1.3 Cache Block Size

The baseline version of Flexicache uses fixed-size cache blocks that are 16 instructions long. After adding all of the chaining optimizations, most of the benchmarks show fairly good performance. Of the remaining benchmarks, three (mpeg2, rasta and twolf) suffer from particularly high miss rates. These benchmarks have working sets (of code, not data) that are too large to fit within the cache block storage area. One factor that exacerbates this problem is the padding added to basic blocks to make them fit the 16-word cache blocks (see Section 4.1.1). A detailed study of this wasted space (see Section 7.3.2) revealed that the padding bloats the program code by approximately 3×. Almost two-thirds of the cache is filled with padding and only one-third holds useful instructions. To reduce the amount of padding needed, the option to use 8-word cache blocks was added to the system.

When using only the chaining optimizations, changing the block size from sixteen to eight words actually hurt performance on all but two benchmarks (see the first two bars in Figure 7-3). This is due to three different effects. First, because the cache can hold a larger number of the smaller blocks, the size of the bookkeeping data structures increases. This takes space away from block storage. Second, more blocks need to be split (because they are larger than the block size) and very large blocks need to be split twice as many times. Each split introduces at least one extra call to the runtime system. Third, the smaller blocks do not allow as many indirect-jump-chaining comparisons to be inserted. Since each comparison requires three instructions, no more than two will fit in the empty space at the end of an eight-word block. This last factor has the largest negative impact since many benchmarks do best with three or four comparisons and a single additional chain

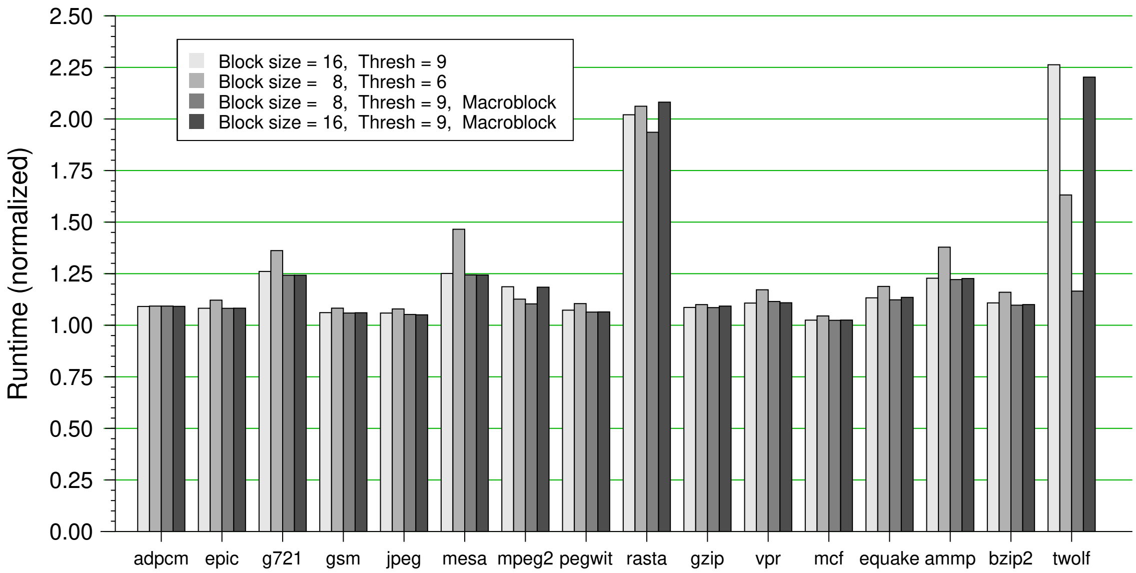


Figure 7-3: Comparison of 8-word and 16-word cache block sizes, with and without macroblocks. Smaller blocks improve performance for applications with larger working sets like mpeg2, rasta and twolf. Macroblocks can be used to eliminate some of the inefficiencies of smaller blocks.

can eliminate thousands of calls to the runtime system. The two benchmarks that show improvement with 8-word blocks (mpeg2 and twolf) have working sets that were slightly larger than the cache, creating a lot of thrashing. In these cases, the reduced padding of 8-word blocks decreased the size of the working set enough so that it could fit within the cache. The time saved by eliminating the thrashing outweighed the time lost due to the above effects.

The macroblock optimization was designed to address the last two effects described above. It reduces the number of splits that have to be made and allows additional empty space to be added to a block for the creation of indirect-jump chains. The third bar in Figure 7-3 shows that macroblocks are very effective in reducing the extra overheads associated with smaller blocks. Essentially, macroblocks allow the system to use larger blocks wherever they are more efficient. The version of the system using 8-word cache blocks and macroblocks produces the best results we've seen on most benchmarks. However, macroblocks have the potential to help even with the larger 16-word blocks. To make sure the good performance was not due to the macroblock feature alone, it was also added to the 16-word cache block system. In this case, the macroblock autoload value was limited to 2 so that both the 8-word and 16-word systems have a maximum macroblock size of 32 instructions. The results are shown in the fourth bar in Figure 7-3 and indicate the value of 8-word blocks. On benchmarks where the working sets were small, the 8-word and 16-word systems perform very similarly. There were only two cases (adpcm and vpr) where the 16-word system performed noticeably better than the 8-word system but the difference was still very small. On the other hand, there were three cases (mpeg2, rasta and twolf) where the 8-word system performed substantially better than the 16-word system. These are the benchmarks with the largest working sets. From this we can conclude that 16-word blocks may provide greater efficiency for some applications with very small working sets but that 8-word blocks are better for most applications. In addition, 8-word blocks can provide huge benefits for applications with very large working sets.

7.1.4 Overall Performance

By combining all of the various optimizations and adjusting the system parameters, good performance was achieved on most benchmarks. Of the sixteen benchmarks, eleven have very good runtime overheads of less than 12% versus a 2-way set-associative hardware I-cache. Put another way, the Flexicache system was able to complete more than two-thirds of the benchmarks using less than 112% of the number of cycles needed by the hardware cache. Of these benchmarks, five of them actually have overheads below 7%. Four additional benchmarks have reasonable overheads between 12% and 25%. Only one application (rasta) continues to have poor performance. Figures 7-4 and 7-5 show the overheads for each benchmark using a variety of I-caching systems. These results show the change in performance as additional optimizations and features are added. All results are measured as cycle counts and normalized to the results from the hardware cache model. The overhead is calculated as the Flexicache runtime divided by the hardware cache runtime, minus one. The unoptimized baseline system is not shown so that additional detail can be seen. For baseline results, see Figure 6-1 in Chapter 6.

Close examination of Figures 7-4 and 7-5 reveals that no single version of the system achieves the best results on all benchmarks. Since the Flexicache is implemented in software and integrated into each program, the programmer is free to choose the version of the system that gives the best performance for a particular application. Table 7.3 gives the actual runtimes for the hardware I-cache model, large memory model and the best software I-cache variant for each benchmark. It also indicates which version of the Flexicache system

Figure 7-4: Performance of each benchmark using several different versions of the Flexicache system. Values on the y-axes are overheads versus the 32 KB hardware cache.

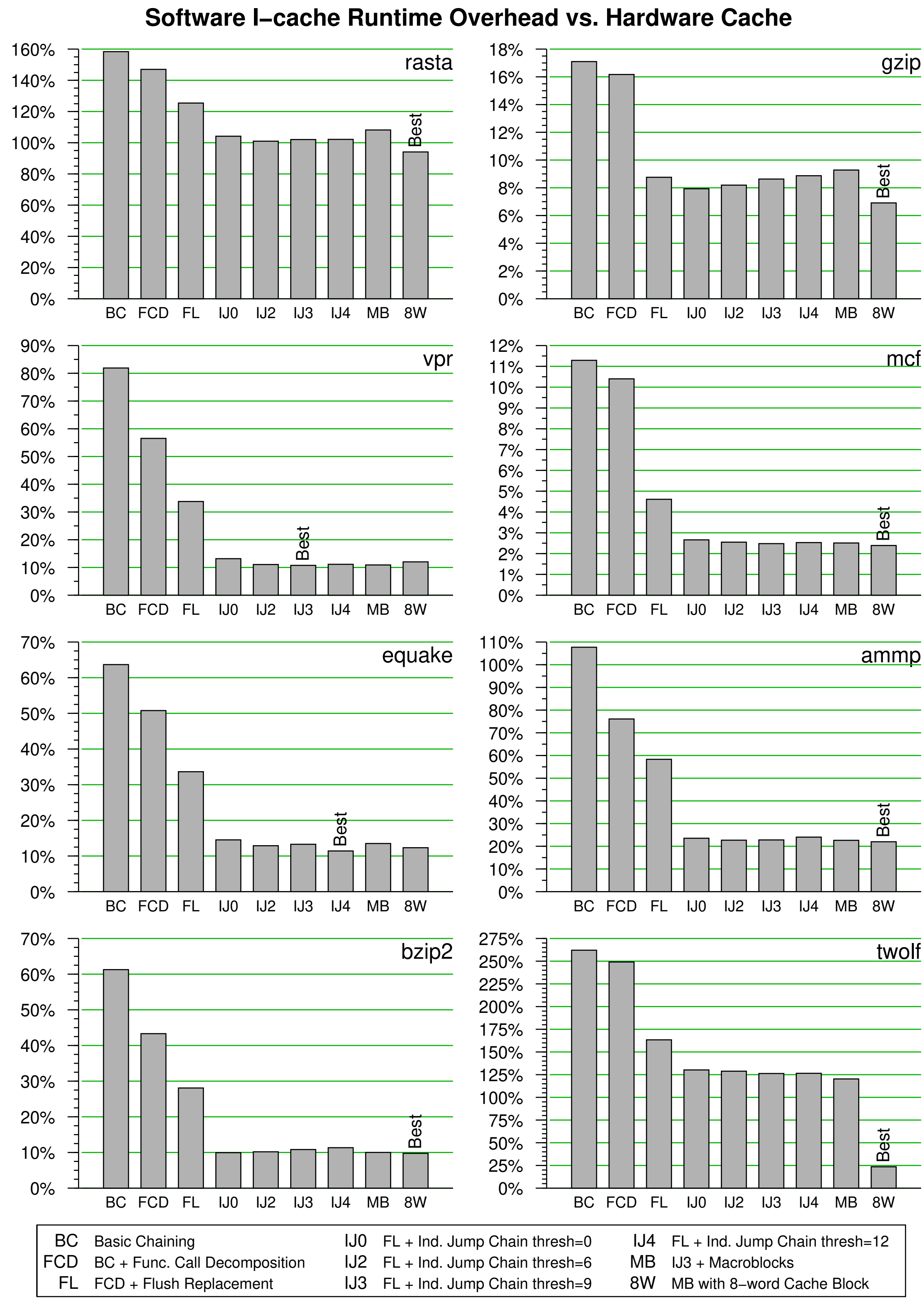


Figure 7-5: Performance of each benchmark using several different versions of the Flexicache system. Values on the y-axes are overheads versus the 32 KB hardware cache.

	Hardware	Large I-mem	Software I-cache		Best Software Config		
Benchmark	Cycles	Cycles	Cycles	Overhead	Blk Size	Thresh	Macro
adpcm	11.2M	11.2M	12.2M	9.0%	16 word	0	No
epic	74.0M	73.9M	80.1M	8.2%	16 word	0	No
g721	367.1M	367.1M	456.1M	24.2%	8 word	9	Yes
gsm	91.8M	91.4M	97.2M	5.9%	8 word	9	Yes
jpeg	45.0M	44.8M	47.2M	5.0%	8 word	9	Yes
mesa	79.8M	—	99.2M	24.3%	16 word	9	Yes
mpeg2	1645.7M	1642.4M	1825.4M	10.9%	8 word	9	Yes
pegwit	69.4M	69.2M	73.8M	6.3%	8 word	9	Yes
rasta	42.6M	40.8M	82.7M	94.1%	8 word	9	Yes
164.gzip	1058.4M	1056.8M	1131.5M	6.9%	8 word	9	Yes
175.vpr	369.4M	346.8M	409.1M	10.8%	16 word	9	No
181.mcf	497.7M	497.4M	509.6M	2.4%	8 word	9	Yes
183.equake	954.2M	955.1M	1063.2M	11.4%	16 word	12	No
188.ammp	64.6M	64.7M	78.8M	22.0%	8 word	9	Yes
256.bzip2	2803.8M	2804.0M	3077.5M	9.8%	8 word	9	Yes
300.twolf	165.7M	144.6M	204.6M	23.5%	8 word	9	Yes

Table 7.3: Run time for Mediabench benchmarks (in processor cycles) using hardware I-cache, large I-mem and software I-caching. "Overhead" is the percentage of extra cycles relative to the hardware version. The final three columns (cache block size, indirect-jump-chaining threshold and use of macroblocks) indicate the combination of parameters that gave the best performance in our experiments.

produced those results. All of the best variants include all of the chaining optimizations and LR spill code rescheduling. They differ in cache block size, empty-space threshold for indirect-jump chaining and the use of macroblocks. Because there are many possible different combinations of system parameter values and optimizations, not all combinations could be evaluated. It is likely that better results could be achieved on some benchmarks; however, we do not expect any dramatically improved results without further optimizations or enhancements.

Note that, in most cases, the hardware cache performance is very close to the idealized large I-mem performance. This leaves little opportunity for the software cache to take advantage of its higher associativity. The exceptions are vpr and twolf where additional profiling indicates that Flexicache does indeed generate significantly fewer DRAM requests than the hardware cache. With additional optimization or with longer DRAM access latencies, Flexicache has the potential to outperform a hardware cache on these benchmarks.

7.1.5 Sources of Remaining Overhead

To guide future system improvements, a study of the remaining overhead was undertaken. The first step was to identify and categorize the potential sources of overhead. In this situation, we define overhead as any cycles beyond those that would be needed to execute the application using the large I-mem model in BTL. Using this model, the application is loaded into the I-mem in its entirety before it is run. The measured runtime does not include the loading phase, only the time needed to execute the application itself. Therefore, overhead includes time spent: in the runtime system code, waiting for fetches from DRAM, and in extra instructions inserted into the user code, either by the rewriter or at runtime. After identifying the sources of overhead, the impact of each one was measured or estimated for each benchmark.

There are two broad categories of places where overhead can occur: the runtime system and the user code. The runtime system consists of the entry point routines, hash table lookup routine, miss handler, etc. Any time spent in the runtime system is considered overhead since it is not a part of the original application and is not needed when using the large I-mem model. Runtime system overhead is separated into three components: front-end, hit handler and miss handler. The front-end component includes the entry point routines and hash table lookup. The hit handler includes code for creating chains and restoring the temporary registers and interrupt state that was saved in the entry point routines. The miss handler performs the same tasks as the hit handler but also fetches blocks from DRAM and updates bookkeeping data structures. Time spent waiting for DRAM to respond is included in the miss handler total. All of the runtime system overhead can be directly measured in the simulator.

"User code" refers to the application that is being run. Any cycles not spent in the runtime system are spent in the user code. Overhead is introduced into the user code primarily through instructions inserted or modified by the rewriter. However, instructions can also be inserted by the runtime system as with indirect-jump chains. Below is a list of the possible sources of user code overhead. Within each section, sources are listed roughly in order from largest to smallest impact.

1) Introduced by rewriter

   (a) Jumps inserted after branch instructions to handle the fall-through control path.
   (b) Jumps inserted at the end of non-branching basic blocks to handle the fall-through control path.
   (c) Jumps inserted to split large basic blocks.
   (d) Link address storage when function calls are decomposed. Three instructions per function call.
   (e) LR spill stores: One instruction per write of LR, may stall if instruction writing LR has latency greater than one.
   (f) LR spill loads: One instruction per read of LR, may cause instruction that reads LR to stall for up to four cycles.
   (g) Instruction to place destination address in $at register before an indirect jump. One instruction per indirect jump.
   (h) Instruction to pass destination to runtime system for indirect function calls. One instruction per indirect function call.

2) Introduced by runtime system

   (a) Address comparisons for indirect-jump chaining.
   (b) Extra data cache misses due to cache pollution.

These overheads are generally more difficult to measure than the runtime overhead because they are caused by instructions that are mixed together with the original user code. For example, an LR spill load (1f above) may add as many as 5 cycles of overhead. However, all but the first cycle will occur as a stall on the user instruction that tries to use the value that is loaded. As another example, after processing by the rewriter, there is no way to distinguish between a jump from the original user code, one inserted to handle a fall-through (1b), and one inserted to split a block (1c).

On the other hand, some of the overheads can be precisely measured or calculated. A sophisticated profiling routine is used to keep track of the dynamically created indirect-jump chains (2a) and count the cycles spent executing them. Jumps that are inserted after branch instructions (1a) are the only instructions that are turned into calls to the *entry2* entry point. Therefore the overhead can be calculated by counting the number of calls to *entry2*. However, when using chaining, the inserted jump will continue to use a cycle, even though the call to the runtime system has been removed. To determine to correct overhead, the number of calls from a non-chaining version of the I-caching system must be used. This same technique is used to calculate overheads 1d, 1g and 1h. The data cache pollution overhead (2b) can be calculated by counting the number of cache misses in user code and multiplying by the time required to handle a miss (34 cycles).

Finally, the LR spill overheads (1e and 1f) are tricky to measure but can be estimated due to the limited and predictable usage of LR. Before implementing LR spill code rescheduling, spill instructions are inserted such that each load will take one cycle to execute and cause four stall cycles and each store will take one cycle and cause two stall cycles. Using profiling, the number of spill loads and stores can be counted and the total overhead can be calculated. LR spill rescheduling removes some of this overhead but cannot always remove all stalls. Therefore, the remaining overhead is estimated by subtracting the improvement seen in the example shown in Figure 6-13. This result should be correct unless macroblocks are enabled, in which case it will be slightly too high.

Figure 7-6 shows the breakdown of the various overheads for each benchmark. These results are collected from the best performing version of the system for each benchmark (shown in Table 7.3). Note that the bars in this graph show what fraction of the remaining overhead is due to each source, not the absolute values of those overheads. The total overhead for each benchmark (taken from Table 7.3) is shown at the top of each set of bars for reference. Some of the bars in the graph aggregate the overheads from more than one of the sources described above. LR spill stores (1e) and loads (1f) have been combined to simplify the graph. The "Other" bar includes both fall-through jumps (1b) and block-split jumps (1c) because they are impossible to differentiate in the current system. It also includes the indirect-function-call setup (1h) because this source was too small to warrant its own bar. In all cases, this overhead is less than 0.3%.

For some benchmarks, the stack of bars is larger than 100%. This occurs when a source has a negative overhead, *i.e.*, the software system executes some portion of the program faster than the large I-mem. When this occurs, the negative overhead is shown as a bar below the 0% line. The positive bars will sum to more than 100% because the total must be 100% when the negative overheads are added in. Negative overheads can occur with either the indirect-jump-chaining source or the data-cache-pollution source.

With indirect-jump chaining, if the first comparison in the sequence succeeds, it takes only three cycles to transfer control to the new block. In the original unmodified program, an indirect jump takes four cycles to execute. Therefore, if the indirect-jump address prescreening is very effective, the total time spent executing comparisons can actually be

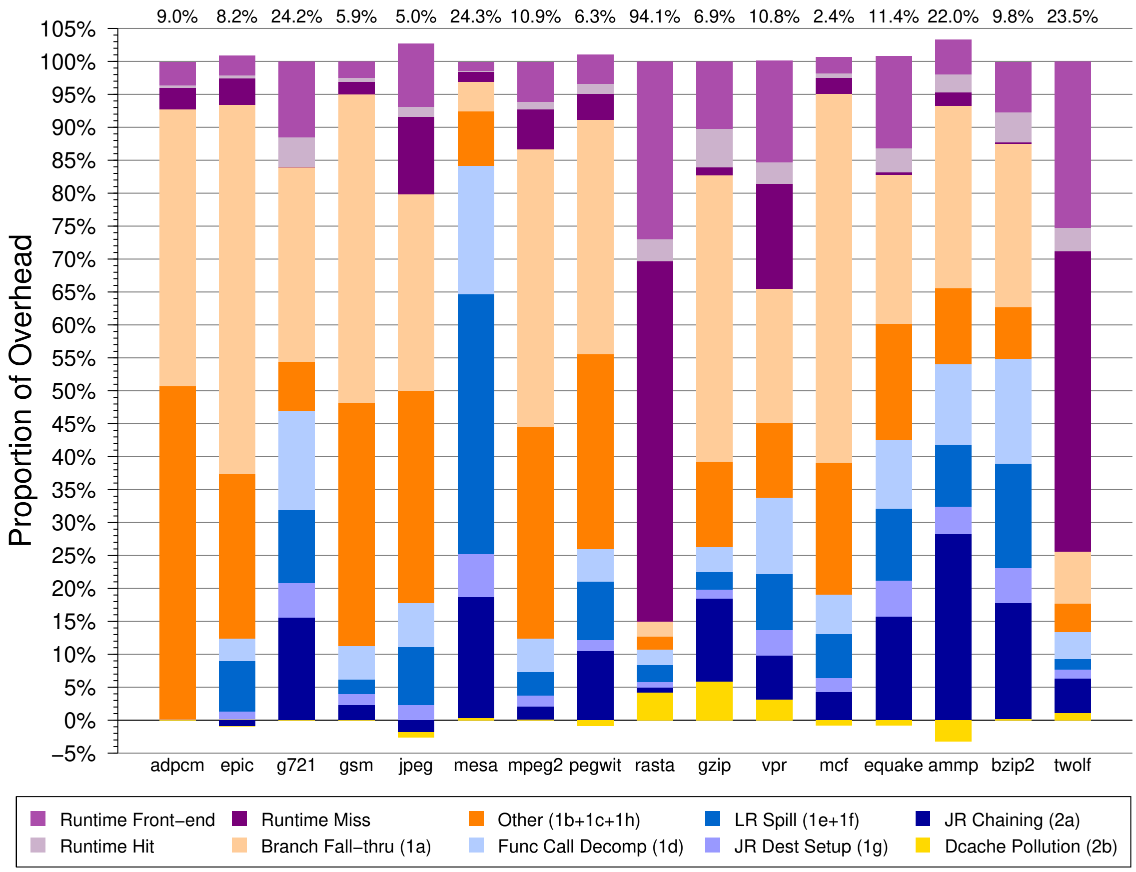


Figure 7-6: Breakdown of the remaining overhead by source for each benchmark. The value above each bar is the total overhead from Table 7.3. Bars extending below 0% represent tasks that the software system performed faster than the large I-mem. When there are negative bars, the sum of the positive bars will be greater than 100% by the same amount.

less than the time spent executing the original indirect-jump instructions, resulting in a negative overhead.

Although the runtime system's use of the data cache usually creates extra cache misses in the user code, it sometimes results in fewer cache misses. This may be due to a different alignment of data or a different initial state of the data cache in the software-cached version of the program. These conditions result from the additional data added to the program by the rewriter. In either case, the reduction in cache misses decreases the time spent in user code and thereby creates a negative overhead. Note that this overhead source only includes data cache misses in the user code. The time required to process any data cache misses that occur in the runtime system is attributed to the runtime overhead component.

As shown in Figure 7-6, the overheads in different benchmarks come from very different mixtures of sources. Even benchmarks with similar total overheads can have very different patterns. For example, g721, mesa, ammp and twolf all have about the same overhead. G721 and ammp derive their overhead from a balanced mixture of sources. However, mesa gets almost all of its overhead from sources related to function calls while twolf spends most of its extra time in the runtime system. Clearly, there is no single area to focus on for future optimizations that will produce dramatic improvements on all benchmarks.

In most of the benchmarks that already have good performance, the remaining overhead is dominated by inserted fall-through and block-splitting jumps. Further improving these benchmarks will require eliminating these jumps, possibly by using larger blocks and/or placing multiple basic blocks within a single cache block. Another group of benchmarks is heavily influenced by the extensive use of function calls. There are several overheads in Flexicache related to function calls, especially functions that call other functions. These include: function call decomposition (1d), LR spill code (1e and 1f), indirect-jump-destination setup (1g) and indirect-jump chaining (2a). These benchmarks could be improved with more sophisticated LR spill code rescheduling, different schemes for choosing and managing the indirect-jump-chaining addresses or minor changes to the hardware. Finally, applications like rasta, twolf and (to a lesser extent) vpr and jpeg spend a lot of time in the runtime system, particularly handling cache misses. Optimizations to target these applications should look at further reducing padding, reducing data structure size and other techniques to squeeze more useful code into the cache and reduce the number of misses.

7.2 Energy Consumption

Energy consumption is rapidly becoming a primary consideration in every processor design. Embedded processors have long sought to minimize power consumption as they are frequently used in portable, battery-operated devices. Reducing energy consumption results in longer battery life or smaller, lighter batteries. Recently, even general-purpose processor designers have begun to focus on power [33, 32, 34]. This is partly due to the burgeoning laptop market and partly due to the high costs of cooling associated with high-power-density traditional designs. Therefore, a software I-caching system will not be practical if it causes the energy required to complete a task to increase excessively. To evaluate the energy usage of Flexicache, we compare it to a hardware instruction cache.

7.2.1 Methodology

To assess the energy consumption of a software I-cache, a version of BTL that is adapted to work with Wattch [13] is used. Wattch provides a framework for estimating the energy utilization of a processor based on the major power-consuming components: I-cache, D-cache, register file, integer and floating-point ALUs, and clock distribution. The models of the various components are adjusted to roughly approximate the power consumption of the actual Raw processor. These adjustments are based on direct measurements of the Raw chip [51] and analysis of its design. CACTI [95, 76] is used to generate the models for the hardware I-cache and SRAM memory as discussed below. In our models, the 32 KB hardware I-cache accounts for about 25% of the total energy consumed.

7.2.2 Hardware and Software Overheads

Previous studies [63, 13, 36, 98] have shown that instruction caches consume a substantial fraction of a modern processor's power. Data from actual processors [63, 13] as well as power estimation tools [36, 98] indicate that instruction caches typically account for roughly 18% to 33% of the total power consumption. Much of this energy is used for things other than the actual data access that is required. For example, a direct-mapped cache performs a tag access in parallel with the data access and compares the tag to the desired value. Set-associative caches typically access all ways within a set in parallel and then discard the ways whose tags do not match the desired tag. When using the software I-cache with a directly-addressed SRAM memory, instruction fetches incur only the data access cost. Of course, a software cache also expends extra energy in the additional instructions it executes to manage itself.

To better understand the relative sizes of these extra energies, we used CACTI 3.2 [95, 76] to estimate the access energy of several different cache configurations. We modeled direct-mapped and 2-way set associative caches (the most popular types for instruction caches) with sizes ranging from 8 KB to 32 KB. Since CACTI does not generate SRAM-only models, we used a direct-mapped cache model and subtracted the energy for the tag lookup and comparison components, leaving only the address decoding and data access components. The results (shown in Table 7.4) indicate that between 20% and 50% of the energy consumed by the caches would be eliminated when using equally sized SRAMs. Note

	SRAM	Direct-Mapped		2-Way Associative	
Size	Energy	Energy	Overhead	Energy	Overhead
8 KB	0.27673	0.38591	28.3%	0.55652	50.8%
16 KB	0.35856	0.47433	24.4%	0.63450	43.5%
32 KB	0.49692	0.62251	20.2%	0.75742	34.4%

Table 7.4: Energy (in nJ) per read access for SRAM and cache models generated by CACTI 3.2. The "Overhead" column indicates the fraction of the total energy used to access tags or unused ways.

that smaller caches have higher overhead because the tags are larger (given fixed cache line and address sizes).

Combining the data from CACTI with the cache power consumption data from the literature, between 6% and 11% of the total processor power is spent on tags or unused ways in a 32 KB, 2-way set associative cache. This is the difference that would be seen if the hardware I-cache could be magically replaced with an SRAM without changing the instructions executed. However, the extra instructions executed by the software I-caching system increase the total energy required to complete a computation and therefore reduce this difference. (On the other hand, it is also possible that a software I-cache could manage the instruction memory more effectively, thereby reducing the energy expended during misses.) Since energy consumption is approximately proportional to the number of instructions executed, roughly speaking, a software I-cache system could incur an instruction overhead of about 10% versus a hardware cache and still consume less energy for a given task.

7.2.3 Simulation Results

To verify the previous analysis, the version of BTL with Wattch was used to estimate the total energy needed to complete each of the benchmarks with both the hardware and software I-caches. The hardware cache is the same 2-way set-associative cache described above. In our model, the hardware cache accounts for about 25% of total processor power. This is consistent with the values reported for actual processors in the literature [63, 13]. For the software cache case, the model for the cache is replaced with the SRAM model while everything else is left the same.

For each benchmark, the version of the software system that gave the best performance was used. Because extra instructions consume extra energy, the fastest version is also likely to be the most energy-efficient. There are two possible exceptions to this rule. First, some

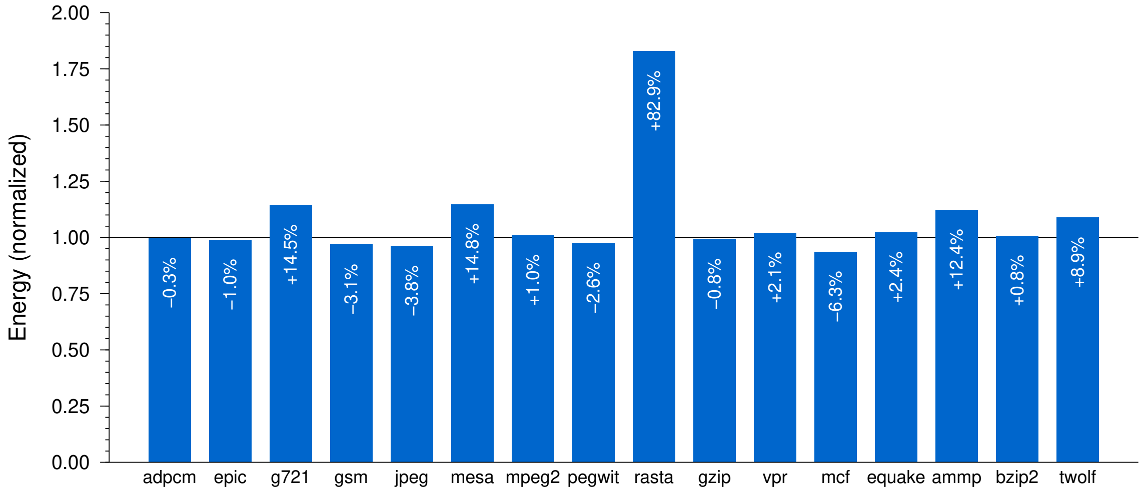


Figure 7-7: Total energy used to complete each benchmark using the software I-cache relative to a 32 KB hardware I-cache. For each benchmark, the highest performance software cache configuration (see Table 7.3) was used. Values less than 1.0 indicate an energy savings compared to the hardware cache.

instructions may consume more energy than others. In particular, instructions that read or write the instruction memory are more expensive than, for example, arithmetic instructions. Therefore, it is possible that a version of the system that makes many dynamic modifications to the cache contents could consume more energy, even though it executes in fewer cycles. Second, this analysis only considers processor energy consumption. Accessing DRAM can require orders of magnitude more energy than accessing an on-chip memory [43]. Therefore, a version of the system that uses sophisticated cache management to reduce the number of DRAM accesses might consume less total system power even though this management requires extra processor power. We leave an analysis of total system energy consumption to future work.

Figure 7-7 shows the amount of energy used by the software I-cache normalized to the amount used by the 32 KB hardware cache. The values on the bars indicate the difference between the hardware and software cases. As expected, the software I-cache is comparable to the hardware for the benchmarks where its instruction overhead is around 10% or less. In fact, on seven of the benchmarks, the software I-cache system actually consumes less energy than a hardware cache would have. All but one of the remaining benchmarks show only modest energy penalties of less than 15%. However, it is also clear from the rasta benchmark that high performance overhead can dramatically increase the energy consumption.

The main focus of this work is adding functionality to processors that lack caching hardware for one reason or another. However, it is worth considering whether one should design processors to use software caching from the beginning. In this case, it is important to consider the possible alternatives. If die area and energy consumption are primary concerns, one might consider using a smaller hardware cache instead of switching to a software cache. By selecting a hardware cache with approximately the same energy per access as the SRAM memory used in the software system, we can compare the performance at a fixed energy point. From Table 7.4 we see that an 8 KB 2-way set-associative cache has comparable access energy to the 32 KB SRAM. Figure 7-8 compares the performance, energy consumption and energy-delay product for an 8 KB, 2-way set-associative hardware cache and a Flexicache system. All results are normalized to the results from the 32 KB hardware cache, as before.

All three graphs in Figure 7-8 show similar results. For most benchmarks, the 8 KB hardware cache produces slightly better performance and energy consumption than the software cache. The hardware cache seems to have a significant advantage in applications where the remaining software overhead has large function-call-related components (*e.g.*, g721, mesa, equake, ammp and bzip2). On the other hand, the software cache shows an advantage on vpr and twolf: applications with large working sets and a lot of overhead due to the runtime system. However, this pattern does not hold for rasta which also has a large working set. The likely explanation is that the software cache is faster for applications with working sets somewhere between 8 KB and 32 KB. For such benchmarks, the software cache's larger capacity (and consequent reduction in miss rate) outweighs the extra management overhead. For smaller applications, the software cache suffers because of user-code overhead. For larger applications, the software cache suffers because it takes longer to handle a miss using software.

7.2.4 Energy Summary

Due to the inherent energy efficiency of simple SRAM instruction memories, software instruction caches can provide modest energy savings compared to hardware caches of similar size. These savings are reduced by the extra instructions executed under software caching. However, a net savings is possible if performance overheads are kept below 9%.

From an energy-delay product point of view, a smaller hardware cache may provide a better trade-off than a software cache. Based on these results, it is hard to recommend

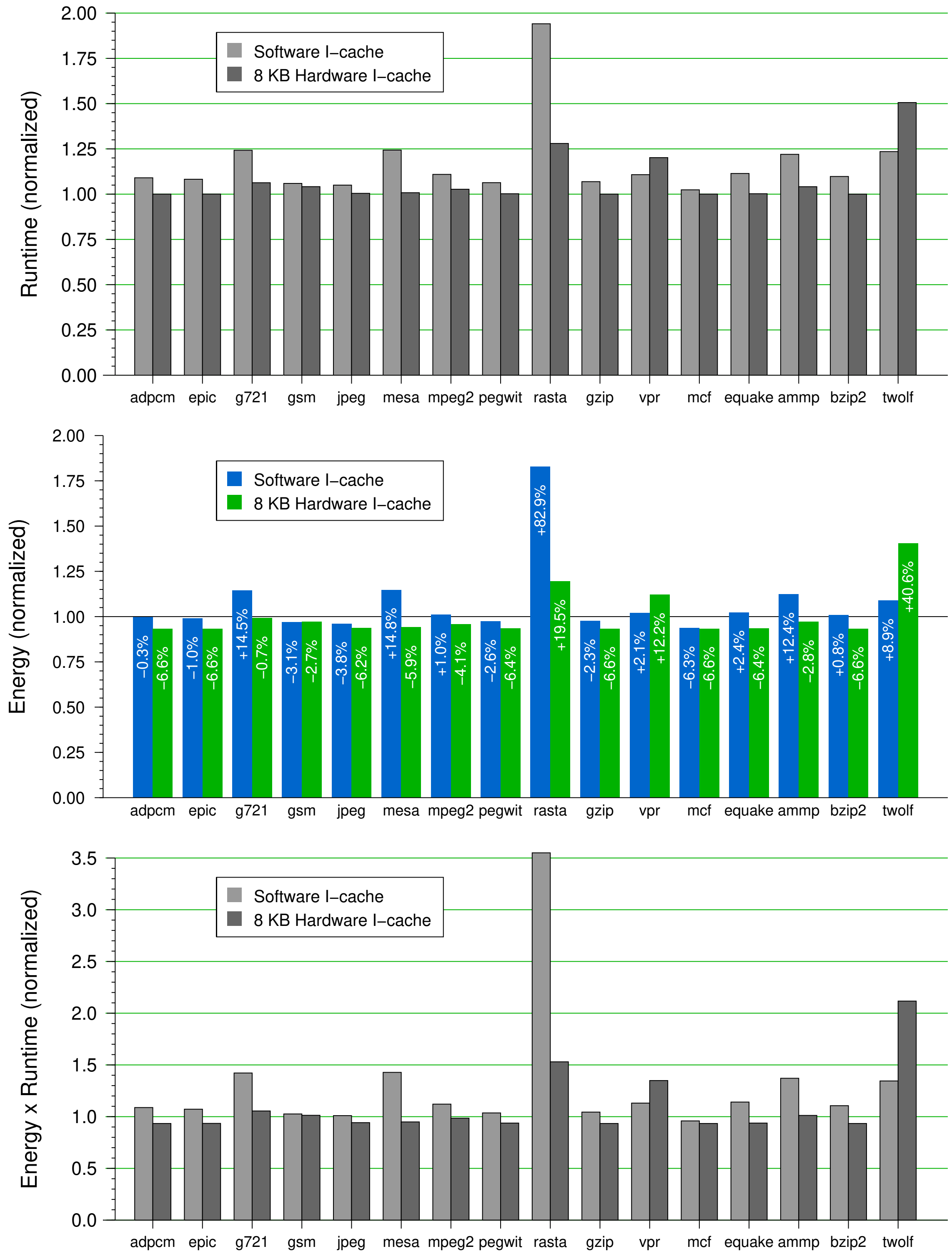


Figure 7-8: From top to bottom: Performance, energy consumption and energy-delay product of an 8 KB 2-way set-associative hardware I-cache compared to the 45 KB software I-cache. All results are normalized to the values from the 32 KB hardware I-cache.

that a processor designer choose a software cache over a hardware cache purely on the basis of performance or power. However, the software cache may still hold advantages in areas that are harder to quantify such as implementation and verification effort as well as timing predictability. Furthermore, future improvements in the software system (see Chapter 10) and hardware mechanisms to assist software caching (see Chapter 8) could negate the small advantage that hardware caches enjoy.

7.3 Other System Characteristics

7.3.1 Hash-Table Conflicts

Section 6.2 discussed the fact that conflicts in the runtime system's hash table can reduce the effectiveness of the cache and negatively impact performance. This section presents data showing that the optimizations that have been implemented are very effective in reducing the number of hash-table conflicts and their resultant misses.

Figure 7-9 shows the number of hash-table conflicts that occur during the execution of each benchmark for various different versions of the system. The results are normalized to the number of conflicts that occur in the unoptimized baseline version of the system. Hash-table conflicts result in cache misses and are therefore analogous to so-called *conflict misses* in a traditional hardware cache. In the baseline system, the simple hash table structure results in a cache that is essentially direct-mapped and therefore has a relatively large number of conflicts. As the first few optimizations are added (up to indirect-jump chaining), the number of chains created increases and the number of conflicts decreases dramatically. For most of the benchmarks, over 75% of the conflicts are eliminated. On five benchmarks (gsm, mesa, mcf, equake and ammp), chaining removes over 99% of conflicts. While it is obvious that chaining should reduce the number of cache *hits*, this data shows that chaining is also capable of reducing the number of cache *misses*.

The four indirect-jump-chaining data-points show that some optimizations can increase the number of hash-table conflicts. Increasing the empty-space threshold allows additional chains to be created but also increases the size of the program code. These two factors tend to have opposite effects on the number of conflicts and the net result is highly dependent on the application. Switching to a smaller cache block size can also increase conflicts. Note that about half of the benchmarks experience an increase in conflicts when changing from

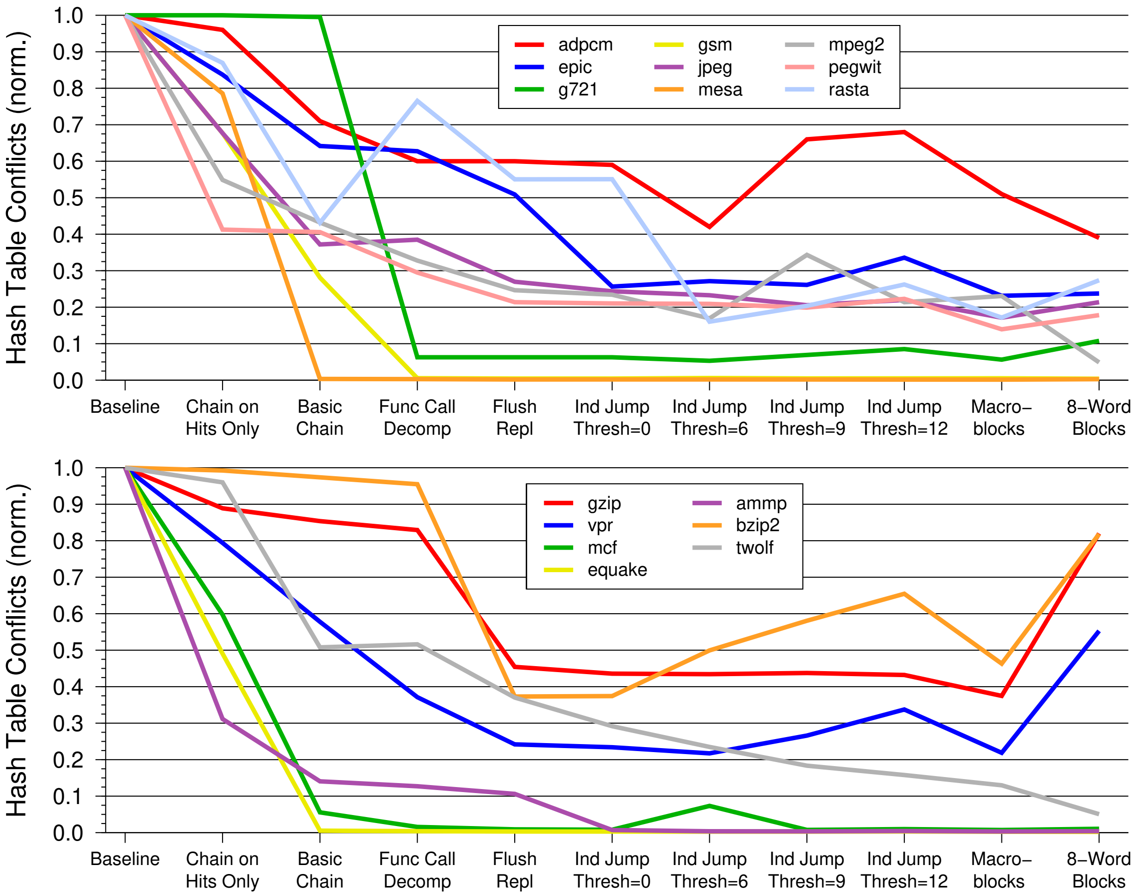


Figure 7-9: Number of hash-table conflicts with different versions of the system. The upper graph shows the Mediabench benchmarks while the lower graph shows the SPEC® benchmarks. The versions of the system are the same as those in Figures 7-4 and 7-5.

the *Macroblocks* version of the system to the *8-Word Blocks* version. Using 8-word cache blocks increases the amount of useful code present in the cache and therefore increases opportunities for chaining. However, it also increases the number of blocks that will fit in the cache. Since the hash table holds entries for all loaded blocks, this increases the probability of a hash collision. One solution is to increase the size of the hash table to maintain the same load factor used with the larger blocks. However, this would rob space from block storage so, in this case, the size of the hash table was kept constant.

One additional point to note is that the macroblock optimization consistently reduces hash-table conflicts but does not accomplish this through additional chaining. Instead, conflicts are reduced because fewer blocks need to be tracked individually. Without macroblocks, large basic blocks are split into multiple cache blocks that are all tracked separately in the hash table. A macroblock replaces these cache blocks with a single block that has only one entry point and therefore requires only a single entry in the hash table. Actually, this is similar to creating chains between the smaller blocks except that, with chains, the initial miss will still create an entry in the hash table that could create conflicts.

7.3.2 Cache Block Padding

As mentioned in Section 7.1.3, the use of a fixed-size cache block tends to bloat the program code due to the padding that must be added to basic blocks. Even if the blocks are somehow stored dense-packed in DRAM (without the actual padding instructions), they will need to be expanded when they are loaded into the I-mem. Thus the padding instructions take up space in the cache that could otherwise have been used to store useful instructions. This code bloat tends to impact programs with large working sets more heavily than it impacts programs with small working sets. This is because smaller working sets (like those found in adpcm, epic, jpeg, gzip, mcf, and bzip2) may still fit within the cache, even after the padding has been added. However, applications with large working sets (such as mpeg2, rasta and twolf) will experience much higher miss rates than would be expected from the unmodified code size.

To better understand the impact of using fixed-size cache blocks, a detailed analysis of the padding inserted into the user's code was performed. To collect the data, the rewriter was modified to make a final pass over the program after all other transformations have been performed. In this pass, the rewriter measures and records the number of useful instructions and the amount of padding in each basic block and calculates statistics for the program as a whole. Note that the number of useful instructions plus the number of padding instructions in each block always add up to the fixed cache block size (either 8 or 16 words).

Table 7.5 summarizes the results collected. All of the data presented (except for the columns labeled "pad") refer to the number of useful (non-padding) instructions in each cache block. Recall that each cache block contains approximately one basic block. Therefore, these numbers are closely related to the sizes of basic blocks in the original program. However, once the basic blocks have been through the rewriter, very large blocks have been split (Section 4.1.1) and additional instructions have been inserted for fall-through paths (Section 4.1.2), LR spilling (Section 4.1.2), and function call decomposition (Section 6.3). Data is presented for versions of the system with 16-word and 8-word cache blocks. In both cases, the table gives statistics for all the blocks in each program as well as just the

	16-Word Cache Blocks								8-Word Cache Blocks							
	All Blocks				Executed Blocks				All Blocks				Executed Blocks			
Benchmark	μ	A	B	pad	μ	A	B	pad	μ	A	B	pad	μ	A	B	pad
adpcm	4.9	4	2	70%	5.2	4	3	67%	4.1	3	2	49%	4.1	4	3	49%
epic	5.0	4	2	68%	5.8	5	3	64%	4.2	4	2	47%	4.5	4	8	44%
g721	4.9	4	2	69%	5.6	4	2	65%	4.1	3	2	49%	4.2	4	8	47%
gsm	5.4	4	2	66%	5.8	5	2	64%	4.3	4	8	46%	4.4	4	8	45%
jpeg	5.4	4	2	66%	6.2	5	3	61%	4.4	4	8	45%	4.6	4	8	42%
mesa	5.6	4	3	65%	6.8	5	3	58%	4.4	4	8	45%	4.9	5	8	39%
mpeg2	5.6	4	2	65%	6.2	5	3	61%	4.5	4	8	43%	4.8	5	8	40%
pegwit	5.9	4	2	63%	7.5	6	16	53%	4.6	4	8	43%	5.2	6	8	35%
rasta	5.4	4	2	66%	5.8	5	3	64%	4.4	4	8	46%	4.5	4	8	43%
164.gzip	5.5	4	3	66%	6.2	5	3	61%	4.5	4	8	44%	4.7	4	8	41%
175.vpr	5.9	5	2	63%	5.9	5	2	63%	4.7	5	8	41%	4.6	4	8	42%
181.mcf	4.8	4	2	70%	5.4	4	3	66%	4.1	4	2	49%	4.3	4	3	46%
183.equake	5.4	4	2	66%	6.3	5	3	61%	4.4	4	8	45%	4.7	5	8	41%
188.ammp	5.8	4	2	64%	5.1	4	2	68%	4.6	4	8	42%	4.2	4	2	48%
256.bzip2	5.8	4	3	64%	6.6	5	3	59%	4.6	4	8	42%	5.0	5	8	38%
300.twolf	5.9	5	2	63%	6.2	5	2	61%	4.8	5	8	40%	4.8	5	8	40%
Average	5.5	4.1	2.2	66%	6.0	4.8	3.5	62%	4.4	4.0	6.5	45%	4.6	4.4	7.0	42%

Table 7.5: Cache block size and padding statistics. On the left are statistics gathered using 16-word blocks and no macroblocks. On the right, 8-word blocks, macroblocks, and an indirect-chaining threshold of 9. Values are given for all blocks in each program as well as just the subset that is actually executed. The columns labeled μ, A, and B indicate the average, median, and mode (respectively) of the number of useful (non-padding) instructions in each cache block. The columns labeled "pad" indicate the fraction of each block, on average, occupied by padding. This is the same as the fraction of the entire program (or executed subset) that is padding.

subset of blocks that are actually fetched and executed in our experiments. For each case, the mean, median, and mode of the number of useful instructions is given. The columns labeled "pad" indicate the fraction of each block, on average, occupied by padding. This is the same as the fraction of the entire program (or executed subset) that is padding.

In general, the numbers of useful instructions in each block are somewhat smaller than we expected. With 16-word cache blocks, the average number of instructions is only 5.5 and the most commonly occurring value (the mode) is only 2 for most benchmarks. Unfortunately, this results in about two-thirds of the final binary being padding. This means that when the blocks are loaded into the I-mem, only one-third of the cache block storage space is actually used to hold useful code. The results are slightly better if we consider only the subset of cache blocks that are actually executed, but they still indicate that more than 60% of the cache space is wasted. The results for the different benchmarks are fairly consistent with each other except for pegwit. Pegwit is unusual because it has two extremely large basic

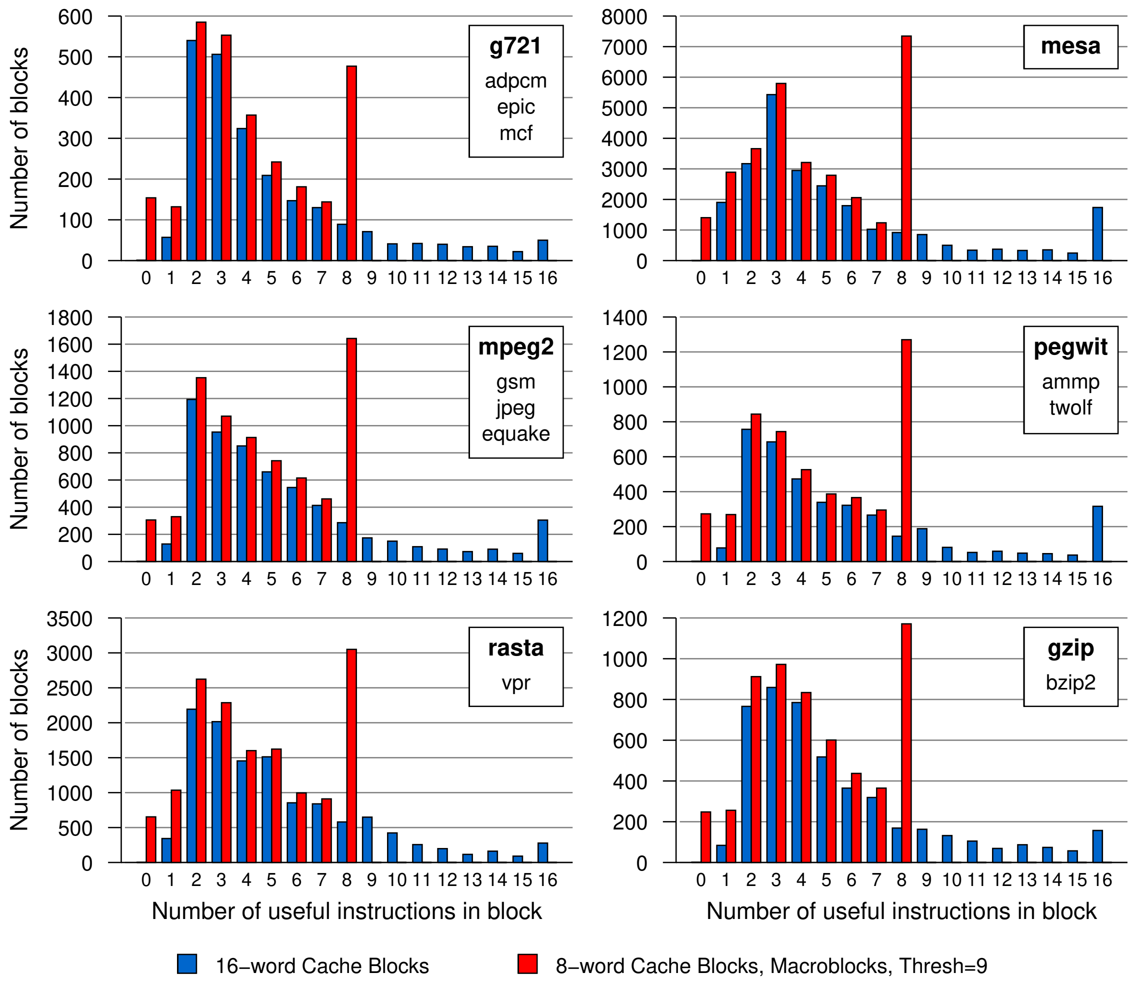


Figure 7-10: Histograms of the number of useful instructions in each block for several representative benchmarks. For each graph, the data shown is from the bold benchmark; the other benchmarks with similar patterns are listed below it. The distributions are heavily weighted toward 2 or 3 instructions but have spikes at the maximum because blocks are clipped to that length.

blocks with 863 instructions in each. These two blocks get broken into many maximally-filled cache blocks, skewing the block size statistics toward the high end.

These results are surprising because previous studies have reported average basic blocks sizes of 6 to 7 instructions [97]. Since the rewriter inserts extra instructions for several different purposes, we expected average block sizes of 8 to 9 instructions. This was part of the motivation for choosing 16-word cache blocks in the initial versions of Flexicache. The difference is probably caused by the splitting of large cache blocks. Although the sizes of basic blocks are heavily weighted toward the low end (Figure 7-10), a long tail of large blocks would bring the average up. In Flexicache, this tail is truncated and the large blocks become multiple small blocks.

The obvious solution to the excessive amount of padding seen with 16-word blocks is to reduce the cache block size to eight words. This successfully reduces the total program padding to about 45% and significantly improves performance on benchmarks with large working sets. In reality, the effective percentage of padding is actually even lower because, in this case, 8-word blocks are used in combination with macroblocks and an empty-space threshold of nine. This causes the rewriter to insert at least one completely empty block after each indirect jump. These empty blocks bring the average number of “useful” instructions down even though they actually serve a useful purpose at runtime. The average number of useful instructions per block is also reduced by the increased splitting that occurs with smaller blocks. However, this statistic is somewhat deceptive since many of these blocks will be loaded together as a single macroblock.

Although reducing the block size effectively reduces the percentage of the cache wasted on padding, it also has some negative consequences. Because a larger number of the smaller blocks will fit in the cache, the size of certain runtime data structures increases, taking space away from block storage. Also, support for macroblocks introduces some additional overhead in the runtime system. (However, without macroblocks, the additional splitting introduces even more overhead.) These factors help explain why some benchmarks perform better with 16-word cache blocks.

7.4 Chapter Summary

This chapter evaluated the Flexicache system from a variety of different perspectives. The first set of evaluations focused primarily on performance (*i.e.*, execution time). First, the performance impact of different replacement policies and cache block sizes was examined. Then, results from all of the various optimizations and system parameters were combined to examine overall performance and find the optimal system configuration for each application. Finally, a detailed analysis of the sources of performance overhead was presented. The second set of evaluations looked at the energy consumption of the Flexicache system versus 32 KB and 8 KB hardware caches. The final set of evaluations examined hash-table conflicts and cache block padding: two system characteristics that have a broad influence on Flexicache's overall behavior and the other results.

The results presented here indicate that a software instruction-caching system can be a valuable tool for embedded systems with explicitly-managed memories. Performance on several of the benchmarks is comparable to both a hardware cache and an idealized infinite I-mem. Most of the benchmarks studied have performance overheads between 2.4% and 12% versus a hardware cache and energy savings of up to 6%. The primary sources of these overheads vary from application to application but extra jumps inserted to handle fall-through paths and factors related to function calls are common culprits. The fact that the caching system is implemented in software allows the programmer to choose the caching scheme best suited to each individual application.

Chapter 8

Hardware Support for Software Instruction Caching

This chapter highlights several architectural features that can improve the performance of a software instruction-caching system. Some of these features are already found in Raw and are exploited by the existing system. Others are suggestions for modifications or additions to future processors that would enable even better results. All of these features are relatively simple additions to most standard processor architectures. Because they are considerably simpler or cheaper than special-purpose caching hardware, they are practical in systems where hardware caches are not. Designers of processors that might make use of software instruction caching should consider including as many of these mechanisms as possible.

8.1 Non-Blocking Memory Access

One area where software caches are at a disadvantage to hardware caches is the length of time it takes to handle a cache miss. Because hardware caches have special-purpose hardware to perform tag checks and bookkeeping, they can typically handle misses very quickly. For example, a miss in the data cache on Raw requires only ten cycles beyond the time it takes to fetch the required block from DRAM. A software cache, on the other hand, may need many normal processor instructions to accomplish the same tasks. In addition, while the optimizations described in Chapter 6 help reduce the number of cache misses, none of them help to speed up misses when they do occur. This is one of the reasons why applications with large working sets (like rasta and twolf) have high overheads compared to a hardware cache.

One way to reduce the extra software overhead is to perform some of the work while waiting for data to be returned from DRAM. Obviously, some tasks must be performed before the DRAM access (*e.g.*, saving registers and interrupt state, and checking for a cache hit) and some must be performed afterward (*e.g.*, storing the returned data in I-mem and restoring the interrupt state). However, tasks like evicting old blocks (including any required unchaining) and updating bookkeeping data structures can be performed at any time after it is determined that a cache miss has occurred. Therefore, we can reduce the total cache miss time by performing them during the long-latency memory access that is required to fetch the new cache block from DRAM. However, since the software cache must perform all of its work on the general-purpose processing core, that core must be free to execute additional instructions during the memory access.

This can be accomplished with either non-blocking memory accesses or a DMA operation. On Raw, normal loads and stores (performed through the data cache) are blocking and therefore stall the entire processor during a DRAM access. However, the programmer has the option of manually creating and sending a request to DRAM by directly accessing the memory network. The processor can then continue to execute instructions while waiting for the reply. When the reply arrives, it is stored in a small FIFO until the processor is ready to consume it. Instructions are then executed to read the data out of the FIFO and store it in I-mem. Using this technique, the software instruction-caching system is able to perform useful bookkeeping work while waiting for DRAM. It would be just as good (or perhaps even better) if the processor had a DMA engine like the one found in the SPE of the Cell processor [35]. In this case, the data would be automatically stored in the I-mem when it arrived, eliminating the need to use normal instructions to store it and allowing additional work (such as restoring of temporary registers) to be performed while waiting.

8.2 Rotate-and-Mask Instructions

Many of the calculations performed by the runtime system involve selecting some of the bits from a number and then multiplying or dividing the result by a power of two. Most often, this occurs when converting an address into an index into the hash or block data tables. The address must be masked to remove some of the low-order bits, shifted right to convert it to a row number in the table and then shifted left to scale by the size of each row. (In practice, this can usually be performed with just two shifts or a mask plus a shift.) To perform these types of calculations more efficiently, Raw includes rotate-and-mask instructions similar to the ones present in the IBM POWER and PowerPC architectures [1]. These instructions perform both a rotate and a mask operation in a single cycle, thus shaving precious cycles off of table index calculations. Since the hash and block data tables are accessed during almost every call to the runtime system, these savings add up.

In addition to the simple rotate-and-mask instruction, Raw (and PowerPC) include a rotate-mask-and-insert instruction. This instruction uses a mask to replace bits in one operand with the bits from a rotated version of a second operand. This instruction replaces a sequence of as many as four instructions: a mask to clear the bits to be replaced in the first operand, another mask to isolate the proper bits in the second operand, a rotate to move the bits in the second operand to the proper position, and an `OR` operation to combine the two words. This instruction comes in very handy when creating chains. It can be used to quickly replace the destination field of a control-flow instruction with a new address. It is also good for filling in the immediate fields of the instructions used for indirect-jump chaining comparisons (see Figure 6-8). In fact, this instruction is generally very useful when performing dynamic optimization by modifying instructions. Between both the table index calculations and the instruction modifications, having rotate-and-mask instructions saves about four to five cycles on every call to the runtime system. This assumes that the processor without rotate-and-mask instructions would at least have flexible mask instructions. If this were not the case, the savings would probably double.

8.3 Specialized Control-Flow Instructions

As mentioned in Section 4.1.2, Raw implements several novel control-flow instructions that are used just for software instruction caching. These instructions (called *conditional jump-and-links*) perform a comparison and, if successful, use an absolute address to jump to a destination while saving the address of the next instruction in the link register. It is easiest to think of these instructions as different versions of the conventional conditional-branch instructions that they are designed to replace. They function in exactly the same way as the branches except that they specify their destination using an absolute address rather

Original Program Code	Modified Without Conditional Jump−and−Link	Modified Using Conditional Jump−and−Link

<pre>L1: op $x,$y,$z
    op $x,$y,$z
    beq $3,$4, L2</pre>
	<pre>L1: op $x,$y,$z
	      op $x,$y,$z
	      beq $3,$4, L1a
	      jal r.entry2
	L1a: jal r.entry1</pre>
		<pre>L1: op $x,$y,$z
		    op $x,$y,$z
		    jeql $3,$4, r.entry1
		    jal r.entry2</pre>

(a)	(b)	(c)

Figure 8-1: Examples of different ways to modify branch instructions. Part (b) shows how the branch would be modified using only conventional MIPS instructions. Part (c) shows the improved modification possible with conditional jump-and-link instructions. r.entry1 and r.entry2 refer to entry points in the runtime system.

than a PC-relative one and they have the additional task of saving the link address. If the comparison returns `false`, the instruction simply falls through to the next instruction and does not modify the link register. Although the behavior of these instructions may seem complex, they are actually very easy to add to a typical processor design because all of the required datapaths should already exist. These instructions are used to replace conditional branches so we know that the appropriate comparison logic is already present. Most RISC processors also have some form of jump instruction that takes an absolute address and some type of instruction that saves the link address. Therefore, only the instruction decoding and some control logic needs to be changed to add the new instructions.

While conditional jump-and-link instructions are not absolutely necessary for software instruction caching, they help reduce overhead in several small ways. The key advantage of these instructions is that they allow the rewriter to directly replace conditional branches with a single instruction. As discussed in Section 4.1.2, it is important that calls to the runtime system use absolute addresses so that they can be placed anywhere in the I-mem without having to fix-up their destination fields. These calls also need to communicate their location to the runtime system by saving the link address. The original branch instructions have neither of these properties and must therefore be modified. Conditional jump-and-link instructions are designed to replace branches and provide the required properties.

Without conditional jump-and-link instructions, the original branch would need to be replaced with two instructions: one to check the condition and another to perform the call to the runtime system (see Figure 8-1(b)). This would add an extra cycle of overhead to all runtime system calls corresponding to taken branches. If these calls are chained by modifying the `jal` instruction, the extra cycle of overhead will be incurred every time the branch is taken. This can be avoided by modifying the destination of the branch instead. However, the branch requires a PC-relative destination which would take longer to calculate than an absolute destination. Therefore, the cost of creating these chains would go up. Plus, since the chaining routine would now need to treat different calls in different ways, the cost of creating all the other chains would go up as well. Finally, the extra instructions would take up additional space in the cache, reducing the amount of useful code that can be stored. These factors would have only a minor impact on smaller programs that suffer few misses. However, the larger programs such as mpeg2, rasta, vpr and twolf could be impacted significantly.

As convenient as conditional jump-and-link instructions are, they could be even better. Based on the analysis of the remaining overhead in Section 7.1.5 it is clear that some applications suffer significant performance degradation due to the way that Flexicache uses the link register (LR). Because every call to the runtime system uses LR to communicate its location, the user program's version of LR must be stored in a spill location in switch memory. The extra instructions required to perform this spilling create the LR-spill overhead shown previously in Figure 7-6. However, it also means that when function calls are decomposed, an extra instruction must be used to place the link address in the spill location. Plus, an extra instruction is needed for most indirect jumps to retrieve the value from the spill location. Thus, the use of LR is also responsible for one-third of the "function call decomposition" and nearly all of the "indirect jump setup" overheads. In the case of mesa, these three factors account for more than 50% of the remaining overhead.

All of this overhead could be eliminated if Flexicache could avoid using LR for its own needs. The first step to achieving this is to modify the conditional jump-and-link instructions so that they save the link address in a dedicated, special-purpose register rather than the general link register. (This is exactly the same mechanism used by many processors to save the resume point when an interrupt occurs.) The runtime system can then retrieve the value from the special-purpose register when it is needed. Ideally, the conditional jump would store its own address in the special-purpose register, rather than the address of the next instruction. This would save an instruction in the runtime system and offset the extra instruction needed to fetch the address from the special-purpose register. Currently, the

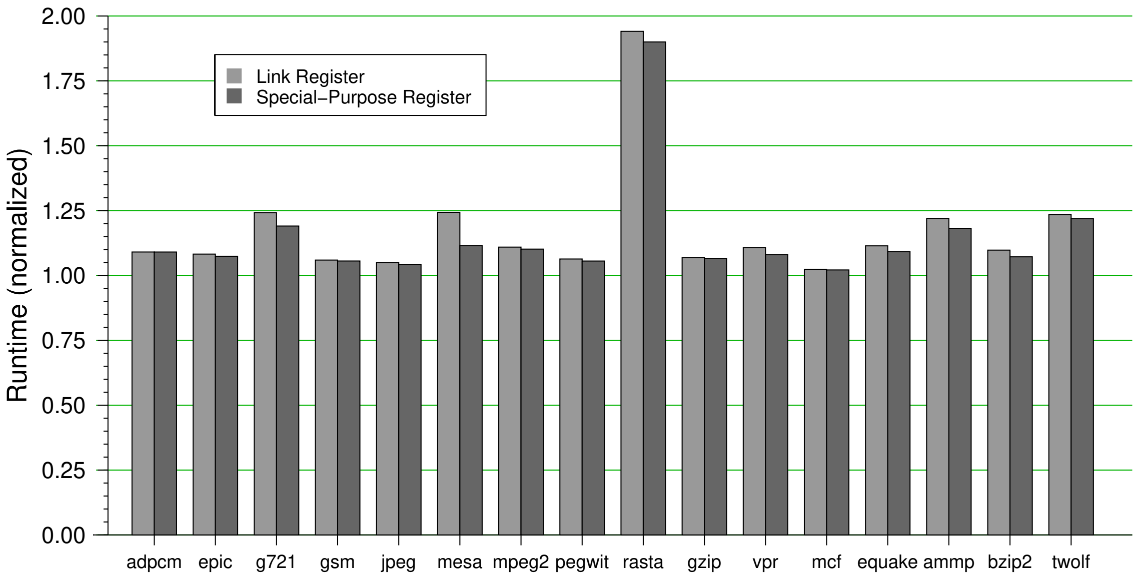


Figure 8-2: Estimated performance improvement using a special-purpose register instead of LR to communicate a call's address to the runtime system. Savings is due to the elimination of LR spilling and is estimated based on the previous overhead analysis.

runtime also uses LR when jumping back into user code. Thus, the processor would also need an instruction that jumps to an address stored in a special-purpose register, just like the interrupt handler return instructions `dret` and `eret`.

These two changes would allow for much more efficient operation. Intuitively, the current system constantly keeps LR saved in switch memory just in case the register is needed for a call to the runtime system. This incurs overhead even when calls have been chained and no longer need to use LR. Using the new instructions, overhead would only be incurred when a call to the runtime actually occurs. Figure 8-2 shows the estimated improvement of storing the link address in a special-purpose register instead of LR. There is significant improvement in the benchmarks that perform many function calls: g721, mesa, rasta, vpr, equake, ammp and bzip2. Note that these were essentially the poorest performing benchmarks and that this optimization has brought them more in-line with the other benchmarks.

8.4 Multiple-Access Instruction Memory

Many low-cost embedded processors use a single-ported SRAM for their instruction memory. This saves area but means that instruction fetch must be stalled for a cycle whenever data is explicitly read from or written to the I-mem. Thus, instruction-memory loads and stores effectively occupy the pipeline for two cycles. For example, on Raw it takes 16 cycles to store an 8-word cache block in the I-mem. Besides storing cache blocks, the I-mem is also accessed for reading and writing bookkeeping data structures, creating chains, and spilling LR. Therefore, these extra pipeline stalls can add significant time to runtime system calls and even modified user code.

Switching to a dual-ported SRAM would allow instruction fetching to continue during explicit I-mem accesses, thereby eliminating the stalls. However, dual-ported SRAMs are significantly larger (and therefore more costly) than single-ported SRAMs. A cheaper alternative employed by processors like Scale [54] and Cell [35] is to alter the aspect ratio of the instruction memory so that multiple instructions are fetched simultaneously. These instructions can then be placed in a separate, very small buffer and executed from there. While instructions are being executed from the buffer, the instruction memory is available for other load and store operations. Although some stalls may still occur if too many loads or stores are executed close together, this solution can hide some of the accesses and is inexpensive. In fact, wide-access memories are frequently faster and more energy efficient than memories with extremely narrow interfaces.

8.5 Dedicated Register Space

Another significant source of overhead in the runtime system is the need to save and restore all of the registers that the runtime system uses. Because calls to the runtime can occur at any point in the original program, every register must be treated as a *callee-saved* register. In other words, the runtime system is responsible for restoring the value of every register that it changes. (The only exception is LR which the runtime system is free to corrupt because the user-program-visible state of this register is actually kept in the spill location. See Section 4.1.2.)

The entry point, hash table lookup and hit handler portions of the runtime system use five registers (in addition to LR). In the event of a cache miss, the miss handler uses an additional two registers. Register values are stored in dedicated spots in switch memory to speed access and because the runtime system can not be sure that the user program has set up a stack. Each read or write to the switch memory takes one cycle because the runtime system code has been carefully optimized to avoid any stalls. Thus, between 10 and 14 cycles are used to save and restore registers for every call to the runtime system. In the case of a cache hit, this represents about 25% of the time needed for the entire call.

This overhead could be eliminated if the runtime system had access to dedicated registers that did not need to be preserved across calls. Note that it is not sufficient to use general-purpose temporary registers if the user program is also allowed to use them. This is because the software instruction cache's "calls" can occur at any time and do not adhere to normal calling conventions. A call to the runtime system could occur in the middle of a computation using the temporary registers. The runtime system can only avoid saving and restoring registers if they are dedicated solely to its use.

There are several possible ways to provide these registers. If a processor contains a sufficiently large number of architectural registers, it may be acceptable to simply reserve a few of them for the software instruction-caching system. Only programs that adhere to this restriction would be cacheable. If non-compliant programs needed to be handled, the rewriter could perform register re-allocation to free up the reserved registers. However, this might require the rewriter to insert extra register spills into the user code. This is undesirable since it would add overhead to the program even after the calls to the runtime system have been removed by chaining (and therefore the spill would no longer be needed).

Another potential way to provide dedicated registers is with a secondary register file. In essence, the runtime system would be treated as a second thread with an independent register state. When the runtime system is invoked, an instruction would be executed to switch the processor from the normal register file to the secondary file. All operations performed by the runtime system would then affect the secondary file but leave the primary file untouched. When the call to the runtime system is complete, the processor would be switched back to the primary file for use by the user program.

A third alternative is to use a shadow register file like the one found in IBM's S/390 G5 microprocessor [77]. In a single cycle, the entire contents of the register file can be copied to or from the shadow register file. This would allow the runtime system to perform all of its saves and restores in only two cycles rather than fourteen. However, as with the dual-ported instruction memory, these schemes can be costly. Duplicating the register file may not be an efficient use of space in some processors.

8.6 Chapter Summary

This chapter described five simple hardware mechanisms that can improve the performance of a software instruction-caching system. These mechanisms are generally much cheaper and easier to implement than a traditional hardware cache, making them practical for low-cost systems. *Non-blocking memory access* allows the processor to perform useful cache management bookkeeping while waiting for data to be returned from DRAM. *Rotate-and-mask instructions* combine two common operations into a single efficient instruction. *Specialized control-flow instructions* allow most calls to the runtime system to be performed with a single instruction, thereby reducing code bloat and runtime overhead. A *dual-ported instruction memory* would allow faster instruction storage, bookkeeping and chaining but would require considerable extra area. Finally, *dedicated register space* would allow the runtime system to avoid costly saving and restoring of user register values on every call.

Chapter 9

Related Work

The Flexicache system builds on previous work in several different types of systems including: software virtual memory, dynamic binary translators, and software-managed caches. This chapter presents an overview of these different systems with examples of each. Similarities and differences between these systems and Flexicache are highlighted.

9.1 Virtual Memory

Flexicache has its roots in the early virtual memory work of the 1950's and 60's [22]. Systems from that period were usually built with a primary core memory that was directly accessible by the processor and a secondary disk or drum storage. To run programs that were larger than the primary memory, code would have to be brought in from the secondary storage at the appropriate times [12]. This arrangement is similar to the abstract processor model presented in Section 2.1 and the architectures of many embedded processors that use explicitly-managed memories today.

Before the development of hardware caches, the dominant strategies for implementing virtual instruction memory were overlays [71, 78] and segmentation [67]. Both of these systems work similarly to Flexicache in that they divide up the program into blocks and then load blocks from the drum as they are needed. However, overlays typically use a much larger granularity than Flexicache, placing one or more entire procedures in each block. Segmentation systems can have large or small granularities but divide up a program into uniform blocks without regard to program structure. Both of these methods can result in large amounts of extra code being loaded when a particular piece is needed. In addition, overlay systems have rigid constraints regarding which blocks can be loaded simultaneously. Each overlay is assigned to a level based on its position in the program call graph. When a new overlay is loaded, it replaces the previous overlay of the same level, even if there is other unused space in memory. Flexicache is far more flexible because it can store any subset of the program basic blocks at any given time, allowing it to adapt to the dynamic needs of the program. Finally, neither of these types of systems attempted to use optimizations like chaining because it was believed that the overhead required to create chains would outweigh the benefit. For a modern treatment of overlays, see the compiler for the Cell SPE [28].

9.2 Dynamic Binary Translators

Systems that manipulate or modify the instructions of a program as it is executing form a large class of applications called *dynamic binary translators*. This class includes simulators/emulators, dynamic code generators, and run-time optimizers. These applications have many similarities to Flexicache because both types of systems work with program instructions and form a virtualization layer between a program and the hardware on which it is running.

Simulators and emulators (such as Shade [19, 20], Embra [97], DAISY [26], DELI [24], and Wentzlaff's virtual architectures [93]) attempt to imitate the operation of a processor or system while running on a different system. With emulators, the goal is usually backward- or cross-compatibility (*i.e.*, the ability to run programs from an older or incompatible system). Simulators are typically used to test or profile applications before they are used on the intended target system. This may be done because the target system is not available or it may be done to execute the application in a controlled environment where detailed analysis can be performed or side-effects can be sandboxed. Fast, modern simulators and emulators use dynamic binary translation techniques to translate the machine code of the original program into machine code for the host computer. This code can then be executed natively by the host processor. To amortize translation costs, blocks of translated code are stored in a *translation cache* (sometimes called a *code cache*) and executed from there. The translation cache is, in essence, a virtual instruction cache.

Dynamic code generators (including VCODE [29] and just-in-time compilers [3, 55]) produce sequences of machine instructions on-demand. Typically, the code to generate is specified by calls to an API or pre-compiled bytecode. Generating code at runtime allows the code to specialized to particular runtime conditions or allows the distributed binary to be platform-independent. As with simulators, generated code fragments are usually stored in a code cache so that they can be reused.

Runtime optimizers (*e.g.*, Mojo [17], Dynamo [7], and DynamoRIO [14, 15, 52]) attempt to modify a program while it is running to increase performance or security. These systems usually form code traces composed of several basic blocks and then apply optimizations to them. To detect traces and monitor the program's execution, control-flow instructions may need to be modified to jump to handler routines. Once the traces have been optimized, they are placed in a translation cache for execution.

Many of the mechanisms used to manage translation caches are similar to the mechanisms used by Flexicache. In fact, Shade and Embra were the original inspiration for the earlier work [61, 62] that evolved into the Flexicache system. In most dynamic binary translators, processed blocks are placed in the cache with their control-flow instructions modified to jump to a runtime routine. Shade introduced the use of chaining to eliminate calls to the runtime system and proposed limiting chaining to simplify unchaining [20]. However, it made no attempt to optimize indirect jumps. Most code-caching systems since have found that chaining is essential to achieving good performance and that indirect jumps limit performance significantly if left unoptimized [97, 26, 49, 15]. Embra [97] made the first attempts to chain indirect jumps but its technique allowed for only one destination at a time (equivalent to having only a single address comparison in Flexicache). The Flexicache technique is very similar to the one introduced in DAISY [26]. Indirect-jump optimization is important and difficult enough that others have even proposed hardware mechanisms to address it [49].

However, there are two major differences between these systems and a software instruction cache. First, the translation cache is typically stored in the main memory of the host computer and therefore can usually be sized to accommodate all but the very largest programs [20, 15]. Because of this, most dynamic binary translation systems only need to deal with their caches becoming full on an infrequent basis. On the other hand, instruction caches are usually much smaller than the program they are trying to run and may fill frequently. Therefore, the replacement policy and mechanisms have a greater influence on overall performance in a software I-cache. In addition, infrequent cache overflows allow dynamic binary translators to amortize the costs of optimizations over a longer period of time. Mechanisms for software I-caches need to be fairly lightweight since the code that is loaded may not be in the cache for very long.

The second major difference is that an instruction cache only loads code into SRAM while a dynamic translator must also translate it. In some cases the translation is limited to inserting code snippets for profiling or debugging. In others, the original instructions may need to be analyzed and reimplemented on a completely different architecture. The extra overhead for translating a piece of code is substantial. Translation overheads can range from 10 instructions (Shade [19]) to 4315 instructions (DAISY [26]), on average, for *each instruction* of the source program. Because of this extra overhead, cache misses are much more expensive in a translation system than they are in an instruction-caching system. As a result, the two types of systems may make different trade-offs between cache management overhead and miss rate. For example, in the Flexicache system, switching from the FIFO replacement policy to the Flush policy resulted in better overall performance, despite the fact that it increased the cache miss rate. The same might not be true if cache misses were two to three orders of magnitude more expensive (as they are in translation systems).

Despite these differences, the core mechanisms in dynamic binary translators and software instruction caches are very similar. The Flexicache system could be used as a platform on which to build an emulator, virtual machine or dynamic optimizer.

9.3 Software Caches

Earlier work in software-managed caches has dealt primarily with hardware caches utilizing software miss handlers [18, 47, 37] or level-two caches [60, 37]. However, there have been some more aggressive designs that rely solely on software for primary caches [31, 65]. There are also other techniques make automatic use of scratchpad memories [8, 88, 6] but they generally optimize only select portions of a program rather than providing a complete caching solution.

Systems such as the VMP multiprocessor [18] and softvm [47] have hardware caches but employ software cache miss handlers. VMP and softvm have hardware to check tags and handle cache hits but they fire special interrupts to invoke software handlers on a miss. This approach allows for some customization of cache behavior to a particular program and eliminates some of the cache hardware (*i.e.*, the cache miss state machine). However, the tag storage and comparison structures are still required. Furthermore, VMP uses a local memory, separate from the cache, to store the miss handler routines. This means that there is a static partitioning of the total local memory between the cache and the miss handler. With Flexicache, there is a single unified memory, allowing for a flexible partitioning of resources.

RAMPage [60] is an example of a system where some part of the memory hierarchy is under software control but not the lowest-level cache. In the case of RAMPage, the level-one caches (both data and instruction) use conventional hardware designs but the unified level-two cache is managed by software. If there is a miss in the level-two cache, the software system is invoked to fetch the needed data from DRAM. This design permits some customization for an individual program and eliminates the level-two cache hardware, but still requires expensive hardware for the level-one cache.

The *indirect index cache* (IIC) [37] allows blocks to be loaded anywhere in the cache and uses a hash table to keep track of them as Flexicache does. However, it implements this hash table (and the lookups within it) in hardware and only invokes software for cache misses. This results in hardware that is even more complex that a traditional hardware cache. Further, it does not attempt to run the software miss handler on the primary CPU but, instead, assumes a tightly coupled coprocessor. Finally, as with RAMPage, the IIC is intended as a level-two cache, leaving the performance-critical level-one cache to a conventional hardware design.

SoftCache [45, 30, 31] seems to be the only modern system (besides Flexicache) to use software-only caching for its lowest-level instruction memory. Softcache and Flexicache both divide up the original program into basic blocks and modify control-flow instructions that leave each block. They also use similar hash tables to keep track of blocks and chaining to eliminate hash-table lookups when possible. However, an important difference is that SoftCache assumes a client-server model where an embedded client machine is supported by a large remote server. The client uses software-only caching to save area and power but relies on the server to perform all of the complex cache-management operations. The Flexicache system executes entirely on the core CPU and requires no external support. Interestingly, the SoftCache authors conclude that software caching is not practical for single-processor, workstation-class machines like the one used by Flexicache. This thesis demonstrates that aggressive optimization and careful system design can make software instruction caching practical on a broader range of machines.

HotPages [65] (and the follow-on FlexCache [64]) is the data cache equivalent to Flexicache. Although hardware cache designers typically implement instruction and data caches very similarly, the differences in the way that instructions and data are used require somewhat different software implementations. The baseline functionality for the two systems is very similar. However, the differences arise when trying to optimize away calls to the runtime system. While program code has a simple, easily analyzable structure (a control-flow graph), data has a more complex patterns and is less predictable. HotPages uses pointer analysis to analyze memory accesses and then uses optimized checks when it thinks the requested data is likely to be in the cache already. However, it is still forced to do some sort of check for nearly all requests. Because instruction streams are more predictable, a software instruction cache is frequently able to remove the check completely (see Section 6.2).

With the growing prevalence of scratchpad memories [8] in embedded processors, several other techniques have been proposed to make automatic use of them. Some of these techniques focus on data [8, 25, 88] but several use scratchpad memory for code [6, 89, 79, 74, 88]. These techniques use profiling or program analysis to identify pieces of code that are likely to be used multiple times and then either statically map them to the scratchpad or insert code into the program to copy them into the scratchpad before they are used. The assumption here is that instructions are normally fetched and executed directly from an external memory and are only copied to the scratchpad as an optimization. Flexicache can be used for this type of environment but is also applicable in the more challenging situation where no code may be directly executed from the external memory. In this case, one does not have the luxury of picking and choosing the code that is cached but must instead manage all code that is executed. Furthermore, the off-line, profile-driven selection of regions to copy can lead to very poor performance if the dynamic execution of the program is substantially different than expected (due to unusual input data, for example) or if the program exhibits different phases. Flexicache handles all fetch decisions dynamically and can therefore adapt to unusual patterns or phased execution.

Chapter 10

Future Work

Although the Flexicache system is fairly robust and highly-optimized, there are still numerous opportunities to experiment with and improve it. This chapter briefly discusses some of the opportunities that we have identified so far. These include: improved indirect-jump chaining, alternative replacement policies, alternative cache block formation strategies, increased customization to individual programs, and additional hardware mechanisms.

10.1 Indirect-Jump Chaining

Profiling results show that there are still a significant number of indirect jumps that fall through to the runtime system in some benchmarks. These jumps are very costly since they incur the overheads for both the indirect-jump-chaining comparisons and the call to the runtime system. More sophisticated versions of the indirect-jump-chaining optimization might improve these results.

In the current system, addresses are added to the sequence of chains as they are encountered. Once an address is added to the end of the sequence, it cannot be changed until the block is evicted and reloaded from DRAM (clearing the entire sequence). This does not necessarily yield an optimal sequence of address comparisons. Ideally, comparisons should be performed in order from most-frequently-encountered to least-frequently-encountered. Just because an address is encountered first, does not mean that it belongs at the beginning of the list. In fact, the first addresses encountered may be in initialization code and should not be in the sequence of checks at all after that initial phase.

One possible improvement is to make the sequence of address comparisons more dynamic. Ideally, each of the comparisons would be monitored and the ones that succeed most often would be migrated to the beginning of the sequence. In practice, this monitoring is impossible to do without adding extra overhead to each jump. A good compromise might be to reorganize the sequence periodically, perhaps after some number of jumps fall through to the runtime system. Large numbers of calls falling through could be a good indication that the list needs adjustment.

The simplest adjustment would be to clear the entire sequence and start building it again from scratch. This would ensure that no stale, unused addresses remain in the list; however, it would probably also remove some chains that are still needed, thereby causing extra calls to the runtime system until the chains are recreated. Another option would be to rotate the list: removing the first address and shifting the others up. Over time, this would also eliminate unused addresses but might cause fewer extra calls to the runtime system. Finally, a third option would be to add newly encountered addresses to the beginning of the sequence (rather than the end) and remove the oldest address if the maximum sequence length is exceeded. However, this scheme has the potential for serious thrashing if the number of actively used addresses exceeds the maximum sequence length.

Another possible enhancement would take the opposite approach to making dynamic adjustments and use more static analysis instead. The rewriter could be modified to analyze procedure call points and compile lists of possible return addresses for each function. These lists would represent the possible addresses for the indirect jumps that are used as function returns. The rewriter could then use this information to add just the right amount of empty space after each indirect jump (rather than using a single global threshold as it does now). Going one step further, the rewriter could create and insert the sequence of chaining comparisons itself, eliminating the overhead of building the sequence dynamically at runtime. Furthermore, application profiling information could be used to determine the best ordering of addresses within the sequence. If there are a large number of possible return addresses for certain indirect jumps, it would be very inefficient to check for them all sequentially. In this case, the rewriter could build a comparison tree, create chains for only the two or three most frequent addresses, or simply disable chaining for this jump.

10.2 Replacement/Eviction Policy

The ability to use complex replacement policies is one of the key advantages of a software caching system. By taking advantage of the general-purpose processing power that is available, software caches can make intelligent replacement decisions that lead to fewer cache misses. The FIFO and Flush policies used in Flexicache represent two ends of a spectrum: evicting one block at a time versus evicting all blocks in the cache. There is still ample room for experimentation in between these two extremes.

In our earlier work [61] we proposed an organization of the I-mem that we referred to as a *segmented heap*. This scheme would divide the I-mem up into several segments. Newly loaded blocks would then be assigned to a segment, either using a hash function or by filling each segment sequentially. When a new block needs to be stored in a segment that is already full, only that segment would be flushed (as opposed to the entire cache). This scheme would permit unlimited, efficient chaining within a segment and would only require recording (and possibly limiting) chains that cross from one segment to another. Although we have not yet had an opportunity to implement this scheme, we believe that it may combine the low miss rate of the FIFO policy with the improved chaining of the Flush policy and provide better performance than either one of them. Recent studies on similar schemes by Hazelwood *et al.* [38, 40] seem to support this conclusion.

Besides looking at new replacement policies, we feel that the old FIFO policy may merit additional study. Based on the results seen with the Flush policy, it seems clear that the restrictions imposed on chaining under the FIFO policy (Section 6.2) negatively impact performance. However, the FIFO policy shows promise for reducing the number of cache misses. Therefore, given what we now know about the power of chaining, it would be prudent to re-examine those restrictions and investigate methods of easing them. This could be as simple as allocating as extra column in the block data table for unchaining information. Or it might involve dynamically allocating space to allow any number of chains to be tracked (possibly using linked lists). Any method of easing these restrictions will require additional space and time for bookkeeping. However, the benefits of creating more chains may outweigh the increased overhead.

10.3 Cache Block Formation

Based on the padding results presented in Section 7.3.2, it is clear that significant space is still being wasted in the cache. There are two approaches to reducing wasted space that we have identified: eliminating the need for padding and replacing padding with useful code.

The first approach is to eliminate cache-block padding by switching from fixed-size cache blocks to variable-sized cache blocks. Each cache block could then be made exactly the right size for the basic block it contains. The variable-sized blocks would be packed densely in I-mem rather than placed in fixed slots. Although this type of scheme is certainly feasible it would require extensive changes to the operation of the runtime system and rewriter. Because the cache blocks would not be aligned in the I-mem, there would no longer be a fast, efficient way to find the correct row of the block data table from the link address passed to the runtime system. To make matters worse, the number of cache blocks that could fit in the cache would be highly variable, requiring the runtime data structures to be dynamically re-sizable. Thus a completely different scheme for communicating destination addresses to the runtime system would probably be required. It is not immediately obvious whether the reduction in wasted space would be worth the extra complication (and overhead) of using a variable-sized cache block.

The second approach to reducing wasted space is to pack more code into each block instead of adding padding. The simplest way to do this would be to join adjacent basic blocks until they fill a cache block. By loading multiple cache blocks at once, there is the possibility that some of the code that is loaded will never be executed. However, if the alternative is to use padding, it is better to load something that *might* be needed than to load useless `nop` instructions.

There are several ways that basic blocks can be merged. Superblocks [46] are formed by joining basic blocks into single-entry, multiple-exit, non-looping regions of the control-flow graph and may be a good choice. However, it would also be easy for Flexicache to support regions with multiple entry points or with arbitrary internal edges, including loops. Using the macroblock feature, larger pieces of the graph could be loaded as a single unit. Not only does this have the potential to reduce wasted space, it could also eliminate calls to the runtime system. Any edges that begin and end within the same cache block can use relative jump destinations and avoid the runtime system entirely. The effect would be the same as if all the jumps within the block were pre-chained, except that it would also eliminate the extra jump instructions inserted for fall-through paths.

10.4 Customization for Individual Programs

We have only begun to explore the possibilities for customizing cache behavior to the particular needs of an individual program. The ability to pin critical pieces of code in the cache is one form of customization. Another is the ability to use different sizes of cache blocks or turn features like macroblocks on or off. There are additional opportunities for customization that we have not yet explored in detail. For example, the size or organization of the hash table could be changed. Programs that use smaller cache blocks might benefit from devoting more space to the hash table or using a table with two entries per key in order to reduce conflicts. Programs that use larger blocks or make heavy use of macroblocks may be able to reduce the size of the hash table and recover additional space for code storage. The hash function could also be altered for each program to minimize conflicts or thrashing.

At present, each of these adaptations must be selected manually by the programmer and statically integrated into the program. A more intelligent rewriter might be able to select (or at least suggest) appropriate features automatically. This might be based on static analysis or even profiling using representative inputs. Profiling could be used to compare the effects of selecting different options. However, it could also be used to identify pieces of code that are performance-critical or suffer frequent cache misses. These pieces could then be incorporated into a single large cache block or even pinned in the cache to reduce cache misses and power consumption and increase performance. Beyond the rewriter, an enhanced runtime system might even be able to dynamically adjust certain system parameters as the program runs. This might be accomplished by borrowing techniques from dynamic optimizers like those mentioned in Section 9.2.

Chapter 11

Conclusions

This thesis explores the concept of software instruction caching. Software instruction caching provides automatic management of the level-one instruction memory in processors that lack special-purpose cache hardware. These processors are cheaper and more predictable than processors with hardware caches but are traditionally very hard to program because the programmer must manually manage the instruction memory himself. The Flexicache system that we have developed frees the programmer from this burden and delivers programming convenience, performance and energy consumption similar to what would be expected with a hardware cache. It accomplishes this without requiring complex, expensive hardware or giving up the ability to maintain predictable timing on critical portions of real-time programs.

The Flexicache system uses a combination of static and dynamic techniques to achieve excellent performance. A static, off-line preprocessor adds software instruction caching to the user program and prepares it for efficient handling by the runtime system. By performing as much work as possible before runtime, overheads can be kept to a bare minimum during program execution. The highly-optimized runtime system dynamically manages the instruction memory to keep it filled with whatever subset of the program is currently being used. This allows Flexicache to adapt to runtime conditions or program phases better than coarser-grained approaches like overlays.

However, the key to achieving high performance is actually to minimize calls to the runtime system using chaining. In our experience, every possible opportunity to chain should be taken. The extra overhead of creating chains that do not get used is nearly always outweighed by the benefits of the ones that *are* used. In the end, performance will be limited by the calls to the runtime system that cannot be optimized away. Only after the majority of calls have been eliminated will the efficiency of the transformations performed by the preprocessor become critical.

Our results show that, using aggressive optimizations, an entirely software instruction-caching system can provide performance on par with a hardware cache: overhead was between 2.4% and 12% on most of our benchmarks. Flexicache also demonstrated an ability to use less energy than a hardware cache whenever the overhead on a particular benchmark was below 9%. The energy savings was as high as 6.3% on one benchmark. This clearly makes Flexicache a viable option for a developer faced with the daunting task of manually orchestrating code movement on a processor with explicitly-managed memories.

Although our overall experience with Flexicache has been positive, there is one disadvantage to the system that has cropped up many times: debugging. Because Flexicache is integrated into the user's program, it can be very difficult for the user to determine whether a bug is in his code or in Flexicache. It was also very challenging to debug the Flexicache system itself, even using a known-working program. This is because Flexicache implements an abstraction layer that deals with program control-flow. When something goes wrong with this layer, the processor usually winds up jumping to a random, incorrect piece of code. Sometimes, a large amount of incorrect code may be executed before a user-visible error occurs. Sometimes the jump destination is only slightly wrong and the program seems to execute normally but produces incorrect results. The behavior of the program during these times is extremely counter-intuitive and makes it difficult for even a seasoned debugger to narrow down the source of the problem.

We use two techniques to make debugging easier. First, as discussed in Section 4.1.1, basic blocks are padded with instructions that signal an error instead of with `nops`. Since a significant fraction of the cache is filled with padding, this stops the processor immediately for many errant jumps. Second, using the scripting ability of the Raw simulator, we developed a library of different routines to work with I-cached programs. To make it easier for users to debug their programs, we have routines that skip over calls to runtime system during single-stepping. To help debug Flexicache itself, we have developed routines that

1) interpret and display the runtime data structures,
2) skip to the next call to the runtime system,
3) produce logs of all requests received by the runtime,
4) perform sanity checks when new chains are created,
5) watch for accesses to unexpected DRAM addresses, and
6) collect detailed profiles of runtime system behavior.

This thesis and the Flexicache system do not represent a final product but rather a beginning. We believe that there is still significant potential to improve the Flexicache system and achieve even better results. The current system forms a solid base on which to build more sophisticated software instruction-caching systems. Chapter 10 discussed several possible improvements but there are doubtless many others. As mentioned in Chapter 9 there are many similarities between Flexicache and dynamic binary translators (DBTs). This system could become the foundation for a variety of different DBTs by moving the preprocessing phase to runtime and incorporating analysis, emulation or optimization routines into the runtime system.

It is our hope that this project will encourage others to experiment with software caching and other technologies that break down the traditional abstraction layers between hardware and software. It is our belief that many of these layers will need to be modified, perforated, or removed to see significant performance gains in future microprocessors.

Appendix A

The Raw Microprocessor and Systems

This appendix describes the Raw microprocessor and two hardware systems that have been built around it. These systems were used as the platform for the experimental implementation of the Flexicache system developed in the rest of this thesis. Understanding the design of the Raw processor and systems will help the reader to understand the detailed implementation and evaluations presented in Chapters 4 and 7.

Raw is a research microprocessor that was designed and implemented by the Raw research group at MIT and fabricated by IBM. The Raw systems were designed at MIT and implemented in collaboration with the University of Southern California Information Sciences Institute East (USC ISI East [69]). This appendix gives a general overview of the Raw hardware but focuses mainly on features relevant to the Flexicache system. More information on the Raw project can be found in the following sources: [90, 84, 87, 86, 82, 83].

A.1 Raw Microprocessor

The Raw project began in the late 1990's to address the question of how best to utilize the billions of transistors that would be available on a single chip in the coming decades. Throughout the 1980's and 90's, general-purpose processor designs had evolved by attempting to automatically find and exploit increasing amounts of parallelism in sequential programs: first pipelined single-issue processors, then in-order superscalars, and finally out-of-order superscalars. Each generation employed larger and more complex circuits (*e.g.*, highly-

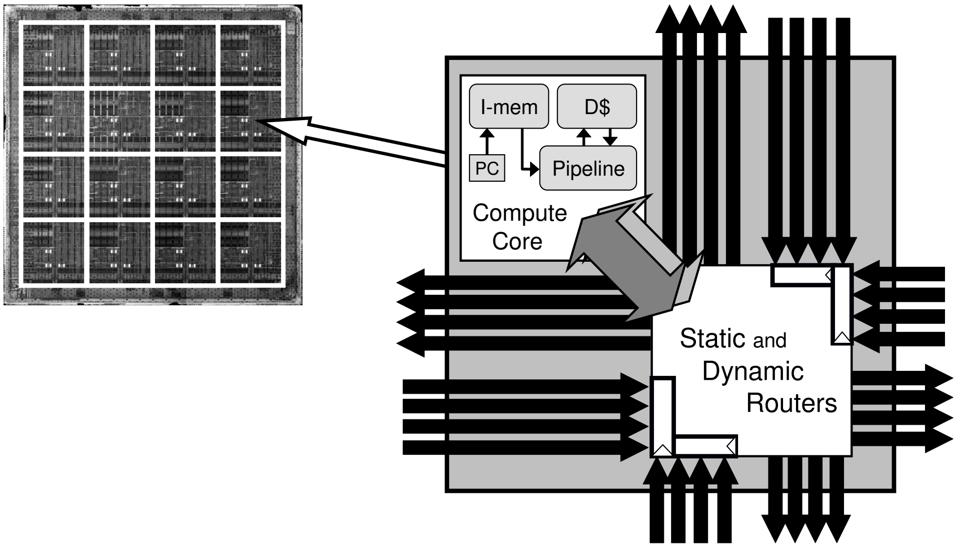


Figure A-1: Overview of the Raw architecture. A Raw processor is composed of a 2-D array of identical tiles. Each tile contains a basic 32-bit computational core and two types of routers to connect it to the neighboring tiles.

ported registers files, massive bypass networks, reorder buffers and load/store queues) to extract additional parallelism from a simple single-threaded program. As clock frequencies increased and wire delay played a larger role in circuit design, it became clear that these large centralized structures would not scale efficiently. Techniques like resource partitioning and super-pipelining allowed for larger, more complex processors but created inefficiencies that resulted in diminishing returns. In addition to performance issues, many of these large structures were power-hungry and very costly to design and verify. The members of the Raw group felt that a new approach to processor design was needed to maintain performance improvement commensurate with Moore's Law in the era of billion-transistor chips.

A.1.1 Design Philosophy and Overview

The Raw architecture addresses the problem of scalability using explicit parallelism and distributed computational elements. A Raw processor is composed of a 2-D array of identical *tiles* as shown in Figure A-1. Each tile contains a simple RISC processing core and two types of communication routers (*static* and *dynamic*) that connect it to the neighboring tiles. The core is kept simple to avoid the inefficiencies of wide-issue superscalars and fit as many tiles as possible on a single chip. As feature sizes shrink, the processor design is scaled by adding additional tiles rather than increasing the complexity of the cores. The communications networks are very low-latency and very tightly integrated into the processing cores to allow multiple tiles to cooperate on a single computation. The networks connect each tile to its four nearest neighbors (to the North, South, East and West) and the network wires are registered at each tile. Thus there are no wires on the chip that are longer than the width of a single tile. This ensures that clock speeds will remain high as the processor is scaled to future process generations.

The Raw architecture takes the approach of making parallelism explicit and exposing the details of the hardware to the programmer (or compiler), rather than using expensive hardware structures to hide the true nature of the processor. For example, a superscalar processor has multiple parallel functional units but uses complex hardware to dynamically assign instructions to the different units at execution time. Raw, on the other hand, exposes its functional units and requires the software to specify a separate instruction stream for each unit. By eliminating large, inefficient, special-purpose structures, Raw is able to provide numerous compact, general-purpose functional units. Because only the essential hardware is retained, Raw is able to squeeze many more functional units onto a single chip than a superscalar can. Each unit and the interconnection between units is exposed to software and individually programmable. This allows the software to allocate units to different tasks as needed, possibly even recreating some of the functionality performed by complex hardware in more traditional processors. Although there may be some inefficiency in performing a particular task in software on a general-purpose unit rather than using special-purpose hardware, the ability to bring so many additional functional units to bear on the overall problem results in a net gain in performance.

The primary reason for eliminating large, centralized structures is scalability. In traditional superscalar processors, the area and delay of structures like bypass networks and register files grow with the square or cube of the issue width. As these structures grow, the wires within them grow to the point that a signal can no longer traverse them in a single clock cycle. When this happens, additional pipeline stages must be added (if possible) or the clock frequency must be reduced. This problem is well recognized in the research community [70, 4, 44, 75, 85] and makes it impractical to scale traditional superscalar processor designs beyond issue-widths of six to eight. In contrast, the Raw architecture can be scaled to an arbitrary number of functional units by adding additional tiles. Since each tile is self-contained and connected only to its nearest neighbors, there are no long wires or

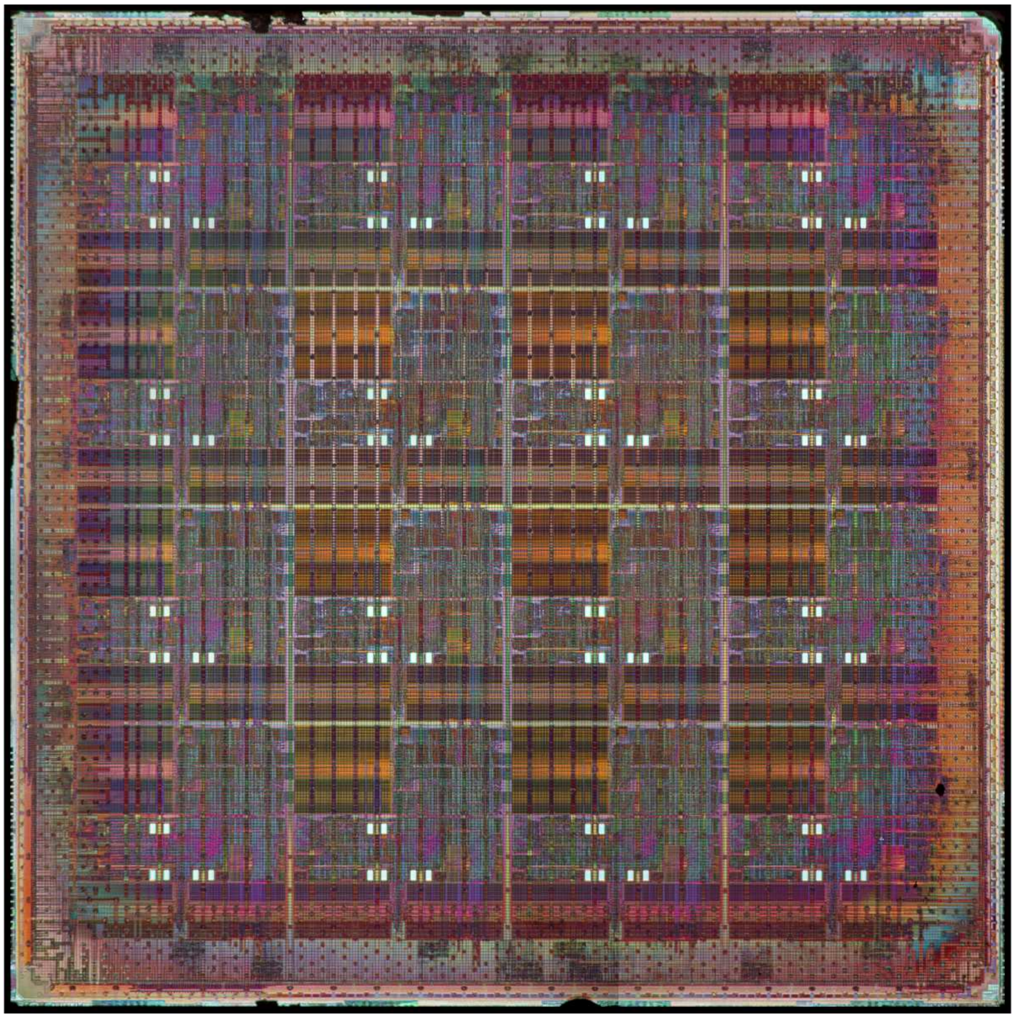


Figure A-2: Die photograph of the prototype 16-tile Raw chip. The regular, repeating pattern of tiles can clearly be seen with the top three layers of metal wiring etched away. Each tile is approximately 4mm on each side. The tiles do not appear identical here due to artifacts of the wafer etching process.

centralized structures that grow inefficiently.

Besides the advantages of scalability and efficient use of die area, a Raw processor is significantly cheaper and easier to build than a monolithic superscalar. This stems from the fact that all of the tiles are identical and relatively simple. The effort required to build a Raw processor is essentially the same as that required to build a single tile. Once a tile has been carefully designed, implemented, and verified, it can be reused as many times as needed to fill the chip area. Whereas commercial superscalar processors require hundreds of engineers and thousands of man-years to implement and verify, our 16-tile Raw prototype was built by a team of less than 10 graduate students in fewer than 30 man-years.

The Raw group has implemented a prototype 16-tile Raw chip (Figure A-2) using IBM's SA-27e ASIC process. This is a 180nm 1.8V CMOS process with 6 layers of copper interconnect. The Raw chip uses an 18.23 × 18.23 mm die and a 1657-pin ceramic column grid array (CCGA) package. Although the 4 × 4 array of tiles would have fit on a 16 × 16 mm die, the larger size was selected to allow the use of a high-pin-count package. The package provides 1152 user-definable signal pins which were used to implement 1080 high speed transceiver logic (HSTL) I/Os for off-chip communication plus a few dozen miscellaneous pins for clock, reset, PLL power and configuration, HSTL reference voltages, manufacturing test, and runtime debugging.

The remainder of this section will describe the various components of the Raw architecture and prototype chip in more detail. Since a Raw chip is composed of a replicated tile, most of this discussion will focus on the contents of a single tile: the computational core, the static network router, and the dynamic network routers. This is appropriate since the Flexicache system essentially treats each tile as an independent processor. However, the final portion of the section will describe the I/O interface of the chip to explain the method used by each tile to access DRAM.

A.1.2 Computational Core

The *computational core* is the real workhorse of a tile. It consists of a 32-bit MIPS-style processing pipeline and accompanying instruction and data memories. In some Raw documents and publications the computational core is referred to as the *main processor* to distinguish it from the *static switch processor* described below. The terms *computational core* and *main processor* are used interchangeably in this document.

Basic Pipeline Architecture

The processing pipeline is an 8-stage single-issue in-order pipeline that contains a 32-bit ALU (Arithmetic Logic Unit), a 2-stage 32-bit integer multiplier, a 4-stage single-precision FPU (Floating Point Unit), and a 32 KB data cache. It supports a MIPS-style ISA (Instruction Set Architecture) that is roughly based on a MIPS R4000 processor [41]. However there are also extensions for custom Raw functions and interfaces to the static and dynamic networks. The pipeline is fully-bypassed and interlocked so that instructions will stall automatically if they are waiting for a result from a long latency instruction.

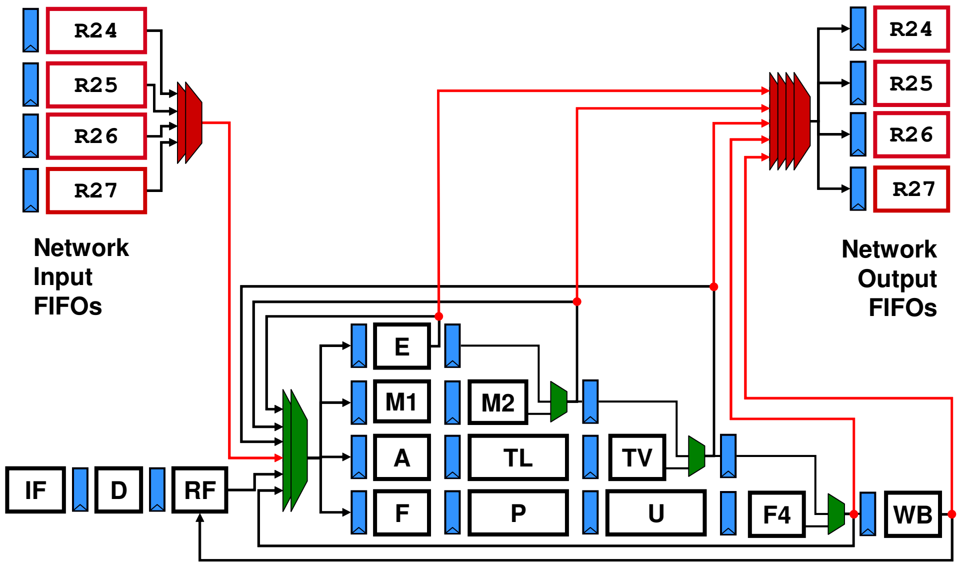


Figure A-3: Block diagram illustrating the pipeline structure of the computational core. This is a single-issue in-order pipeline containing an ALU (block E), multiplier (M1 and M2), 32 KB data cache (A, TL and TV) and FPU (F, P, U and F4). The IF stage contains a 32 KB software-managed instruction memory. The network interface FIFOs are tied directly into the bypass network for extremely efficient communications.

Figure A-3 shows the structure of the entire pipeline. The IF (Instruction Fetch) stage contains a 32 KB software-managed instruction memory (discussed below) and standard program counter logic. Raw uses static branch prediction so all control-flow instructions specify whether they are likely to be taken or not-taken. The A (Address), TL (Tag Lookup), and TV (Tag Verify) blocks form the load/store pipeline. The A stage is used for address generation while the TL and TV stages represent the 32 KB, 2-way set-associative data cache. The actual data and tag memories and the off-chip interface are not shown.

The rest of the pipeline is fairly conventional except for the interfaces to the on-chip networks. Note that the FIFOs coming from and going to the networks are tied directly into the bypass paths of the pipeline. These FIFOs are register-mapped so that any instruction can directly access the networks by simply specifying the correct register number. For example, an `add` instruction that specifies R27 as its destination register will write its result to one of the dynamic networks. Because the output FIFOs are connected to the bypass network (rather than the Write-Back stage), results can be sent as soon as they are produced.

Instruction Memory

Although the computational core contains a traditional data cache, the Raw architects decided to omit a hardware instruction cache and use a simple software-managed instruction memory instead. This decision was based on the goal of simplifying the design of the core to reduce implementation effort and fit more tiles on a single chip. Eliminating the cache tags and control logic saved effort in the design, layout, and verification of the Raw chip. More importantly, it also saved area and removed logic from the chip's critical path. The area saved by eliminating special-purpose structures was used to provide additional general-purpose resources, in keeping with the Raw philosophy. Since the critical path of the Raw processor is already in the IF stage, any additional delay due to cache structures would directly impact the clock frequency or require an additional pipeline stage to be inserted. Raw is not the only processor to deliberately omit caches for the sake of simplifying cores. The SPEs of IBM's Cell processor also use software-managed memories in order to simplify fetch logic and fit more general-purpose resources on each chip [35].

Although it is somewhat incongruous for Raw to have a hardware-managed data cache and a software-managed instruction memory, it provides an excellent opportunity to compare the two types of structures implemented in the same process, using the same libraries, and by the same engineers. Both the instruction memory and the data cache use a 32 KB SRAM to store their data. The layout of the data cache given in Chapter 1 (Figure 1-2) shows that the additional area required to implement a cache instead of a simple SRAM is about 0.94 mm^2. Adding this savings up over all 16 tiles on the Raw prototype results in a total savings of about 15 mm^2 or almost the same area as an entire tile. Thus the use of an instruction memory instead of a cache effectively allows an extra tile to be placed on the chip. Alternatively, if we assume that the total area available for instruction storage is fixed, then the area required for cache management hardware would cut into the space available for instructions themselves. Had Raw used an instruction cache, it would likely have held only 16 KB rather than 32 KB. Besides being larger, the data cache is significantly slower than the instruction memory. As shown in Figure A-3, the data cache takes two pipeline stages (TL and TV) while the instruction memory fits in one (IF). Adding an extra pipeline stage for an instruction cache would increase the branch mispredict penalty and negatively impact application performance.

The computational core uses a Harvard architecture, *i.e.*, independent storage and address spaces for instructions and data. The instruction memory (or *I-mem*) consists of just a 32 KB SRAM memory with a single read/write port. All program counter (PC) values and control-flow destination addresses are used as direct indices into this SRAM. The IF stage of the pipeline simply selects the appropriate next address (*e.g.*, PC+4, a branch destination, the value from the link register, etc.) and feeds it into the I-mem. The values retrieved from the I-mem are then passed directly to the decode stage of the pipeline. Therefore, as far as the rest of the pipeline is concerned, the 32 KB I-mem is the only storage and address space for all instructions. For many DSP-style applications, this amount of storage is sufficient and no virtualization is required. Raw provides optimal efficiency for these applications because no energy, area or pipeline stages are wasted on cache logic that is not needed.

Since the processing pipeline only knows how to fetch instructions from the I-mem, all instructions must be stored there before they can be executed. The Raw ISA provides a special instruction, called `isw` (for *I-mem Store Word*), specifically for this task. The `isw` instruction is analogous to a standard data *store word* (`sw`) instruction and simply stores a 32-bit value from a register in the designated location in the I-mem. This value can be anything but will usually be instructions pulled in from off-chip using one of the communications networks. Because the I-mem has only a single read/write port, the pipeline must stop fetching instructions for one cycle to perform the write. Thus, `isw` effectively takes two cycles to execute.

The Raw ISA also has an `ilw` instruction that performs a read from the I-mem. This instruction is not strictly necessary for normal operation but can be very useful for certain applications. First, it allows self-modifying code to read an instruction out, change some portion of it (*e.g.*, a destination address), and then write the result back for future use. Second, it allows the I-mem to be treated as a scratchpad memory and used to store data. This is particularly helpful for data that needs to be accessed quickly as it avoids the possibility of a cache miss that exists when using the data cache. As with `isw`, `ilw` must stall the fetch stage for one cycle to access the I-mem, thereby requiring two cycles to execute. However, because the I-mem is located relatively far from the rest of the functional units and because it takes an entire cycle to access it, there is a four cycle delay before the retrieved value can be consumed by another instruction. Thus, an instruction that uses the result of an `ilw` may stall for up to four cycles until the value is ready.

Interrupts

One additional set of processor mechanisms worth discussing are the interrupt mechanisms. Interrupts have the potential to influence a software instruction-caching system in two ways. First, interrupt handlers are pieces of code and therefore may need to be cached like any other code. Second, whether interrupt handlers are cached or not, the software caching system needs to ensure that it will operate correctly in the face of interrupts that could occur at any time.

Interrupts on Raw are local to each individual tile. Therefore, the interrupt mechanisms are part of the computational core pipeline, just as they would be with a stand-alone processor. Most of the interrupts on Raw are triggered by internal events within a tile. However, there is also an external interrupt that is triggered by a special message sent to a tile over the MDN. This interrupt (along with an external interrupt controller) allows a tile to be interrupted by external I/O devices. There are two levels of interrupts on Raw: *user* level and *system* level. Both types of interrupts are handled similarly but system-level interrupts have higher priority and can fire even when user-level interrupts are disabled (*e.g.*, while executing a user-level interrupt handler). Special-purpose instructions are used to enable and disable interrupts: `uinton`/`uintoff` affect only user-level interrupts while `inton`/`intoff` affect all interrupts. Individual interrupts within the two levels can be masked using a special control register.

When an interrupt fires, interrupts are disabled, the resume point is saved, and the corresponding interrupt handler is invoked. The resume point is simply the PC of the instruction that would have been executed next had the interrupt not occurred. It is saved in a dedicated status register: EX_PC if the interrupt is system-level and EX_UPC if it is user-level. Once this is done, the interrupt number (between 0 and 7) is converted into an index into the interrupt vector table. Raw uses an unconventional design for its interrupt vector table. The table is stored in the first 128 bytes of the I-mem and contains four words for each interrupt. Rather than simply containing a pointer to an interrupt handler, each entry can contain up to four arbitrary instructions. Instead of using the calculated index to retrieve a pointer, a Raw core transfers the index to the fetch unit and begins executing the instructions in the table entry. Typically, the table entry contains only a jump to the actual interrupt handler. However, the extra storage space can be used by a software caching system to perform a more complex operation when needed (see Section 5.2).

After the interrupt handler has executed, control must be returned to the point where the interrupt occurred. This is done by ending the handler with a `dret` or `eret` instruction. Handlers for user-level interrupts use `dret` while system-level interrupt handlers use `eret`. These instructions perform two operations atomically: they jump to the resume point saved in EX_PC or EX_UPC and they re-enable interrupts for the following instruction. Because interrupts were disabled when the interrupt fired, the entire handler (including the return to normal code) is protected from additional interrupts. Any interrupts that would have occurred during the handler are saved and fire after the `dret` or `eret` is executed.

A.1.3 Static Network Router

The first of the two on-chip communication networks in Raw is the *static network*. The static network is a novel high-bandwidth ultra-low-latency connection between tiles that is crucial for allowing multiple cores to work together on a single computation. However, the current version of the Flexicache system treats each core independently and therefore does not use the static network for performing any communication. Even so, the programs being cached may make use of the static network themselves and there is a potential for interaction between the software caching system and the static network in future work. Therefore, a basic overview of the static network will be presented here but much more information can be found in [84, 82].

The static network is intended to be used for communication that is *static*, *i.e.*, it can be analyzed and routed off-line by the compiler or programmer. For example, consider two cores that have a simple producer/consumer relationship. One core performs some calculations and produces a stream of output values. Those values are then routed to the other core where they are consumed in another calculation. As long as the source and destination tiles are fixed and known ahead of time, the static network routers along the path between the cores can be preprogrammed to provide extremely efficient delivery of operands. The key to the efficiency of the static network is that each router has a predetermined schedule of routes that is executes sequentially. This allows the router to be pipelined and begin preparations for a route before the data has arrived. Because of this and the way that the networks are connected to the computational core's bypass paths, a 32-bit value can be delivered from the output of a functional unit in one tile to the input of a functional unit in a neighboring tile in just three clock cycles. However, because the networks on Raw are registered at each tile, traveling to more distant tiles requires additional hops costing one cycle each.

Because the schedule of static network routes is determined ahead of time, the static network router can be implemented using a conventional (but simplified) processor pipeline architecture. In fact, the static network router is frequently referred to as the *static network processor* or *switch processor* in this and other Raw publications. The routing of different values are encoded as a sequence of instructions for a specialized routing processor. One instruction might route a value from the computational core to the tile to the east. The next instruction might be a through-route that routes a value from the north to the south. Multiple routes can be combined in one instruction as long as the destinations of the routes do not conflict.

The switch processor is essentially a VLIW processor designed to move data around rather than compute with it. Each 64-bit instruction encodes a command for each of the three functional units: a branch/jump unit and two 32-bit 7×7 crossbar units. The branch/jump unit is able to execute standard control-flow operations, simple `MOVE`s between registers and the crossbars, and `nop`s. The crossbars perform the actual network routing and have ports for the neighboring tiles to the north, south, east, and west; the computational core pipeline; the switch processor register file; and the other crossbar. The two crossbars operate on two parallel static networks that connect to the core pipeline through separate register-mapped FIFOs. Data can cross from one network to the other by using the crossbar port that connects to the other crossbar.

The switch processor is organized as a 5-stage pipeline with the first three stages the same as the computational core pipeline. The first stage (IF) contains an SRAM instruction memory (referred to as the *switch memory*) that holds 8192 instructions, just like the core pipeline. However, since the instruction words in the switch processor are 64 bits instead of 32 bits, the total memory capacity is 64 KB instead of 32 KB. The switch processor does not have the ability to manage its own instruction memory. Instead, the computational core has two instructions (`swlw` and `swsw`) that allow it to read and write the switch memory. These instructions work in exactly the same way and have the same latencies as `ilw` and `isw` except that they execute in one less cycle because they do not cause a stall in the fetch unit of the core pipeline. (They *do*, however, cause a stall in the fetch unit of the switch processor.) Thus, the switch memory can be used to store switch processor instructions or as a local scratchpad memory, just like the I-mem.

A.1.4 Dynamic Network Routers

The second type of communication network on Raw is a *dynamic network*. There are actually two completely separate but structurally identical dynamic networks in Raw: the memory dynamic network (MDN) and the general dynamic network (GDN). The dynamic networks are intended to handle traffic which is not statically predictable such as external interrupts, cache misses and data dependent communication patterns. The MDN is used by a limited set of trusted clients (*e.g.*, the data cache hardware, the operating system, hardware devices) to access off-chip resources including memory, the interrupt controller, and I/O devices. The clients must be trusted because the MDN uses strict deadlock-avoidance protocols to guarantee that deadlock will not occur because of overcommitment of buffer space. In contrast, the GDN may be used freely by an application in any way that it sees fit. Since GDN usage is unrestricted, there is the potential for it to deadlock due to overflow of receive buffers. In this event, an interrupt fires which allows a deadlock-recovery routine to clear the GDN using the deadlock-free MDN.

The dynamic networks are packet-based dimension-ordered wormhole-routed networks. Each packet (also called a *message*) on the dynamic network consists of a header word and up to 31 payload words. The header specifies the destination tile, the message type and the message length. As the header makes its way from the source tile to the destination, first by traveling in the X dimension then in the Y dimension, it causes the intermediate routers to create a so-called wormhole for the rest of the message. The routers will continue to forward words along the wormhole path (blocking all other traffic) until the number of words specified in the header have passed. Once the entire message has passed by, the routers are free to examine their input queues and select another message to process. The message type specified in the header is not used in routing the message but is used by the receiver to determine what kind of message it has just received.

As with the static networks, the dynamic network interfaces are tightly integrated into the core pipeline. The FIFOs between the dynamic router and the core pipeline are register-mapped and tied into the bypass paths. The GDN FIFOs are mapped to register 25 and the MDN FIFOs to register 27. To facilitate programming, the assembler allows the use of symbolic names: `$cgno` and `$cgni` map to R25 while `$cmno` and `$cmni` map to R27. Thus a message can be sent from one computational core to another by simply executing instructions whose destinations are the correct register. For example, to send a message on the GDN, a core writes the header followed by each word of the payload to `$cgno`. To receive a message from the MDN, a core simply executes instructions that use `$cmni` as a source operand. There are no special send or receive instructions, any instruction can interact with the networks using the register mapping. For example, an `isw` instruction can specify `$cmni` as a source operand and a word will be dequeued from the MDN input FIFO and written into the I-mem in a single operation.

The primary purpose of the MDN is to transport data between the computational cores and the external DRAM banks. Because of the deadlock-avoidance protocols that must be adhered to when using it, it is treated as a system-level resource that is off-limits for general application use. The GDN, on the other hand, is intended to be a user-level resource that is reserved for a multi-tile application to use for its own internal communications. The Flexicache system is integrated into the user application but is functionally a system service. Therefore, it uses the MDN for all of its communications and does not use the GDN in any way. This preserves the transparency of the system by avoiding any interaction with the application's use of the GDN.

A.1.5 I/O Interface

The Raw architecture uses an unconventional I/O interface that continues the philosophy of avoiding special-purpose structures and exposing the hardware resources to software. Rather than a single wide memory bus, the Raw processor has many smaller I/O ports that are each general purpose. Each port can be connected to a bank of DRAM, an I/O device or both. Because of this flexibility, a system designer using a Raw chip can allocate I/O bandwidth to different resources as demanded by a particular application. For example, a network router might need massive I/O bandwidth to move data around but only a small amount of memory to store routing tables. On the other hand, a server running large scientific codes would need only basic I/O but would want to maximize memory bandwidth and capacity by placing a DRAM bank on every port.

This flexibility is achieved by simply extending the on-chip communication networks to the pins of the chip. Recall that the tiles are arranged in a grid and the on-chip net-

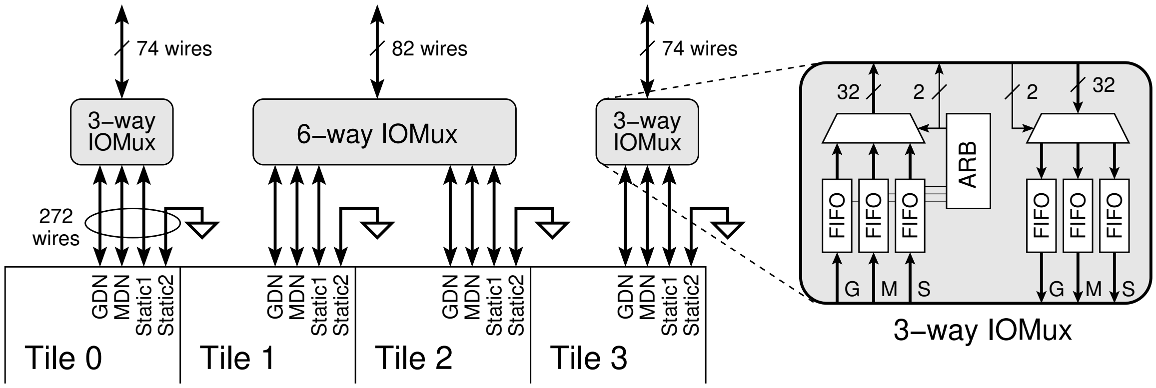


Figure A-4: Diagram illustrating the multiplexing structures used to share package pins among the various on-chip networks. The second static network does not leave the chip. The round-robin arbitrator in each IOMux module sends words alternately from each of the other three networks. The independent buffering prevents traffic on one network from blocking the others. To save additional pins, the networks from middle two tiles on the north and south sides of the chip are multiplexed together onto one set of pins. The enlarged IOMux diagram does not include all control signals.

works form a mesh interconnect. Wherever this mesh reaches the edge of the silicon die, it continues onto the peripherally-placed I/O drivers and then the chip's pins. This scheme has several nice properties. First, it allows multiple Raw chips to be gluelessly connected together to build systems with greater computational power. Second, every core has access to all memory banks and I/O ports because every core can use the networks. Third, off-chip communication can be either dynamically or statically routed. Unpredictable events like cache misses and external interrupts require dynamic I/O. However, some high-speed I/O devices (particularly those that produce or consume continuous streams of data) would benefit from the increased efficiency of a static connection to a particular tile. Fourth, because the mesh reaches the edge of the die at multiple points, there are naturally multiple I/O ports. This allows different ports to be dedicated to different devices, thereby eliminating any arbitration or demultiplexing overhead associated with sharing a port. Of course, for lower-bandwidth devices, it is also possible to establish message formats that allow multiple devices to share a single port.

Pin Multiplexing

Unfortunately, normal microprocessor packaging options do not provide enough pins to simply route all of the network wires off-chip. Each tile-to-tile network link consists of 32 data lines and 2 control lines in each direction. Since there are four different networks (two static and two dynamic) there are a total of 272 wires connected to each side of a tile (Figure A-4). For the prototype Raw chip (consisting of a 4×4 array of tiles), there are sixteen places around the periphery of the array where the networks meet the edge of the chip. Each of these points is referred to as an *I/O port*. With 272 wires in each port, running them all off-chip would require 4352 signal I/O pins. When the implementation of the Raw prototype chip was begun in late 1998, the largest standard package available in IBM's SA-27e process provided 1124 signal I/Os. Although maximum package sizes have increased modestly in recent years, they still fall far short of the required 4352 signal pins.

A combination of two approaches is used on the Raw chip to reduce the I/O requirements. First, one of the two static networks simply terminates at the edges of the chip and does not become part of an I/O port. This was done primarily because the second static network was added to the chip after the implementation was already underway and the designers wanted to avoid making changes to the I/O logic or pinout. Cutting a static network was felt to be an acceptable sacrifice because the compiler or programmer would be able to manually reroute data around the missing links.

Second, the remaining networks are multiplexed onto a shared set of data pins as shown in Figure A-4. This is accomplished by buffering traffic from the three networks at the edge of the tile array and using a round-robin arbitration scheme to select the network that gets to use the pins on each clock cycle. The forward flow-control signals can then be encoded so that two pins are used to indicate which network (Static1, GDN, MDN or none) the data belongs on. (The backwards flow-control signals must remain independent since data can be dequeued from all of the receiving buffers simultaneously.) With multiplexing, each I/O port now requires 32 data pins and 5 control pins in each direction for a total of 1184 pins. Since this is still more than the number of available pins, additional multiplexing is required. To recover the extra pins, the two middle ports on the north and south sides of the chip are combined and share a single set of data pins. This requires a 6-way (rather than 3-way) arbiter and an additional bit for the forward control signals. Using this arrangement, the I/O ports on Raw require 1052 pins, leaving 72 pins for other required uses such as manufacturing test, HSTL reference voltages, clock, and PLL configuration.

The pin multiplexing used on Raw reduces the total available bandwidth at each I/O port but does not change the basic functionality. Each of the networks is buffered independently on the send and receive sides of the multiplexed link. The arbiter is designed so that blocked traffic on one network will not impede the other networks. Therefore, software running on the computational cores may assume that the first static network and both dynamic networks simply continue off the chip. The only user-visible effect of the multiplexing is an increase in latency when using the off-chip link. Even with multiplexing, the total aggregate I/O bandwidth on Raw is extremely high for a general-purpose microprocessor. The I/O logic is designed to run at 300 MHz yielding an aggregate I/O bandwidth of 33.6 GB/s.

DRAM Interface

Raw uses a simple two-level memory hierarchy: on-chip caches backed by off-chip DRAM. Each computational core contains its own hardware-managed data cache and software-managed instruction memory. These caches fetch data from DRAM by sending and receiving messages on the MDN. All transfers to and from DRAM consist of aligned, 8-word blocks corresponding to the line size of the data cache. To request a block from DRAM, a two-word message is sent to the proper DRAM bank. The first word is the packet header and specifies the location of the tile making the request and the *cache line read* message type. The second word is the packet payload and contains the address of the requested block. The MDN transports the message to the correct I/O port where it leaves the chip and is received by a memory controller. The memory controller retrieves the requested data, prepends a header, and injects the message back into the I/O port. Once inside the chip, the message is routed back to the original tile and the payload is delivered into the network input FIFO. Therefore, the latency of a memory access depends in part on the distance the messages must travel on the MDN. Stores are handled similarly to loads except that the original message also contains the data to be stored and the memory controller only sends back an ACK message.

The hardware data cache handles all of this communication behind the scenes. When a cache miss occurs, a state machine in the cache stalls the pipeline and sends a message to the appropriate DRAM bank requesting the missing line. When the reply message returns from the DRAM, the state machine pulls it out of the network input FIFO and stores the line in the cache storage array. Once the line has been stored, the cache retrieves the needed value and unfreezes the pipeline. Thus the cache miss latency is equal to the round-trip latency of a DRAM access plus an additional eight cycles to store the new cache line. This whole process is invisible to the programmer and happens as a side-effect of a load or store that causes a cache miss. However, there is nothing special about the cache's interface to the MDN: it uses exactly the same datapaths and FIFOs that are used with the register-mapped interface. Therefore, software running on the core is able to interact with DRAM in exactly the same way.

Flexicache accesses DRAM in the same manner as the data cache but uses the register-mapped interface to the MDN. When a block of instructions is needed from DRAM, a cache-read message is created and sent. However, unlike the operation of the data cache, this does not cause the pipeline to stall and the processor is free to continue executing instructions. When the instructions return from DRAM, they will simply sit in the processor's input FIFO until instructions are executed to retrieve them. This feature is extremely valuable for software caching as it allows the processor to perform bookkeeping while waiting for DRAM accesses. The only limitation is that the software-caching code must not cause any data cache misses until it has finished retrieving the instructions from the FIFO. If a cache miss were to occur, the cache state machine would mistake the response to the first request for its own data and the two transactions would become corrupted.

A.2 Single-Chip System

A.2.1 Design Goals and Overview

The Single-chip Raw system was the first system designed around the Raw processor and is intended to be a desktop workstation. It contains a single Raw chip, a moderate quantity of DRAM and a variety of different I/O devices and interfaces. Because it is an experimental prototype system designed for a research environment, it also has several features related to debugging and measurement of chip characteristics. The system board adheres to the industry-standard Extended ATX form factor allowing it to fit in an off-the-shelf chassis. Indeed, standard PC parts and connectors were used as much as possible to leverage the maturity and low cost of consumer-grade components. The system uses standard DIMMs for memory, an ATX-12V power supply, a heatsink from a Pentium® 4, a 1.8 V VRM (voltage regulator module) from a Pentium® III, a USB 2.0 connection to a host computer, and PCI slots for adding devices.

Although the Single-chip system board is similar to a PC motherboard in many ways,

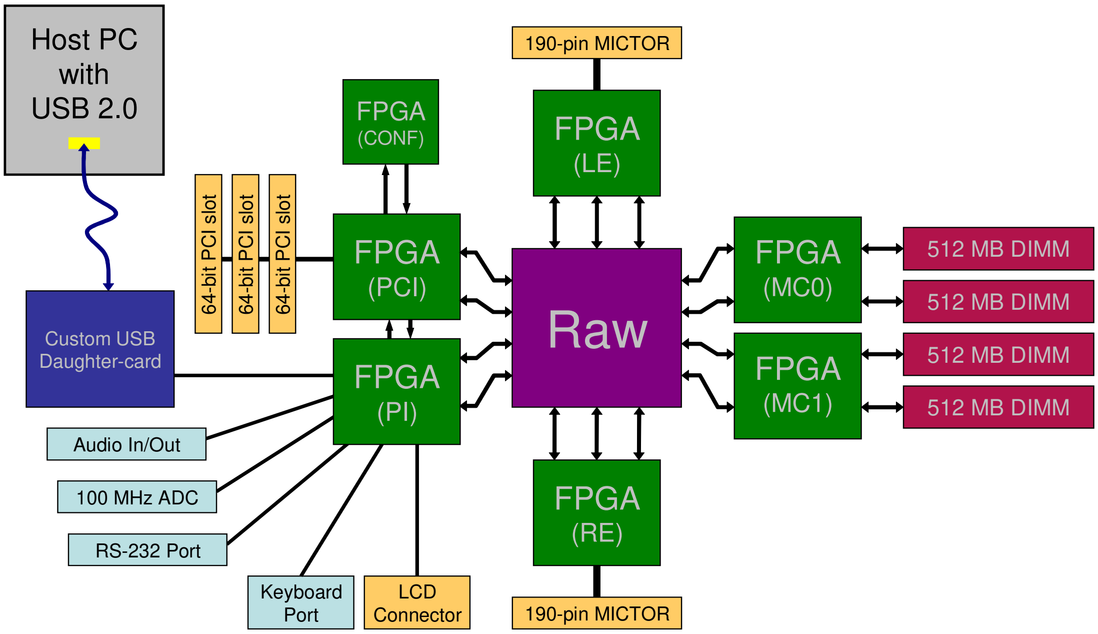


Figure A-5: Overview of the major components of the Single-chip prototype system. The system provides the Raw processor with 2GB of SDRAM and a variety of I/O devices and interfaces. The FPGAs are used to implement device controllers and interface with Raw I/O ports. The USB interface can be used to control the system from a host PC.

in others it is much more sophisticated. The Single-chip system board is a 24-layer PCB (printed circuit board) with thousands of 50-ohm controlled-impedance traces and over 500 buried resistors. These advanced features are required to support the large number of high-speed I/Os on the Raw chip. However, the demands are amplified by some of the features added to the chip for debugging. A version of the board designed for commercial production would probably only require about half as many layers.

Naturally, the design of the system centers around the Raw processor as shown in Figure A-5. Xilinx Virtex-II FPGAs (Field Programmable Gate Arrays) are connected to all four sides of the Raw chip, covering all of the I/O ports. These FPGAs take the place of the custom-built chipsets typically found on PC motherboards. They provide interfaces between the I/O ports on Raw and external devices like DRAM and PCI cards. Implementing the system's chipset using FPGAs allows the interfaces and controllers to be easily modified to fix bugs or add new functionality, vital features for a research prototype. FPGAs provide the flexibility needed for system development and experimentation but cannot match the speeds of normal chipsets implemented as ASICs. Therefore many of the bus and controller frequencies are lower than would be found on a commercial system. The primary subsystems implemented in the FPGAs are: memory, PCI, peripherals, expansion and configuration.

A.2.2 Major Components

Memory Subsystem

The memory subsystem consists of two FPGAs (MC0 and MC1 in Figure A-5) and four industry-standard PC133 SDRAM DIMMs. DIMMs of up to 512 MB are supported, providing a total of 2 GB of external DRAM. Unlike a typical PC, these DIMMs are each on a separate bus and controlled by independent memory controllers. Each of the MC FPGAs contains two memory controllers, each connected to a single DIMM. Each memory controller is, in turn, connected to a single I/O port on Raw. This arrangement allows for greater memory bandwidth and parallelism than a single memory bus would. Different cores in Raw can be accessing different pieces of memory simultaneously as long as they map to different DIMMs. The default memory configuration on the Raw chip assigns each tile a separate 128 MB block of memory in the DIMM located on its row. Thus memory access messages on the MDN only need to travel horizontally and never need to go farther than four hops on the network. With no contention in the network or at the DIMM, the round-trip latency for a memory access from Tile 0 (located farthest from the DIMMs) is approximately 60 clock cycles. A memory controller implemented as an ASIC should reduce this to approximately 30 cycles.

PCI Subsystem

The PCI subsystem consists of one FPGA and three 64-bit PCI slots on a single bus. The FPGA is used to implement a 64-bit/66 MHz PCI controller and interface it to two Raw I/O ports. This allows the use of off-the-shelf expansion cards for things like network interfaces, hard drive controllers and video adapters. A 64/66 PCI implementation was selected over a 32/33 implementation to better match the I/O bandwidth of Raw and support high-bandwidth research cards like the SpectrumWare GuPPI card [11]. Although the PCI controller was completed and tested, the Raw project ended before the necessary drivers and operating system software could be written to make use of any PCI cards.

Peripheral Subsystem

The peripheral subsystem contains an assortment of different integrated I/O devices and connectors. Integrating these devices onto the system board reduces the amount of effort needed to get them working (versus using a commercial PCI card and writing a PCI controller and device drivers) and allows for high-performance, low-overhead interfaces to Raw. The integrated devices include: stereo audio in and out (dual 44 kHz, 16-bit analog-to-digital and digital-to-analog converters), a high-speed (100 MHz) 12-bit analog-to-digital converter, an RS-232 port, a PS/2 keyboard port, a 40x2 character LCD display connector, and a high-performance Teradyne connector for adding a small daughter-card.

The daughter-card connector is used for a USB 2.0 interface card that was also designed and implemented by the Raw group. This card implements a USB *peripheral* interface allowing a Single-chip system to appear as a USB device when connected to a host computer. Software on the host (described in more detail below) allows the user to control the Raw system to load programs and observe results. This is the primary method of using the system until the chipset firmware and operating system software have matured to the point that the system can stand alone and operate as an independent machine.

Expansion Subsystem

The expansion subsystem is actually two separate high-speed interfaces to external boards or devices. It is comprised of two FPGAs (LE and RE) and the 190-pin MICTOR connectors attached to them. The MICTOR connectors are high-density impedance-controlled connectors capable of very high data rates. They can be used with impedance-matched high-speed ribbon cables to transfer huge quantities of data to and from external sources. The FPGAs provide the ability to implement a controller for the external device of to buffer and reformat the data as it is transmitted. These expansion interfaces have been used to collect data from a 1020-node microphone array [92] and to communicate with a software-radio wireless networking board [91].

Configuration Subsystem

The configuration subsystem is responsible for configuring all of the system's FPGAs and setting up the programmable clock generators at system startup. Because FPGAs use volatile storage for their configurations, they must be reprogrammed after every power cycle. This is accomplished using Xilinx's SystemACE solution. The SystemACE solution is a custom chip that reads configuration bits from a standard CompactFlash (CF) card and configures the FPGAs through a JTAG chain that runs through each one. This is extremely convenient for a prototype system as it allows for rapidly switching between different configurations (stored on either the same CF card or separate ones). It also allows all of the configuration bits for the seven large FPGAs to be stored in a convenient removable card. The CF card can be loaded and examined on a regular PC and moved from system to system to test the same bit files on different boards.

Once the FPGAs have been programmed, the CONF FPGA holds the Raw chip and other FPGAs in reset while it configures the programmable clock generators for Raw and the memory subsystem. Having programmable clock generators allows for initial testing and debugging of Raw and the chipset firmware at low speeds. Once the system is working, the clock frequencies can be increased in small increments until the maximum operating frequencies have been found. They also allow for different copies of the board to incorporate different features in the chipset that might influence maximum clock frequency. Finally, varying the clock frequency can sometimes help to expose elusive timing-related bugs. The CONF FPGA is also connected to an 8 MB Flash memory that can be used to store an initial bootup program for Raw.

Power Supply

Power for the Single-chip system comes from a standard ATX-12V power supply designed for personal computers. However, the Single-chip system requires several different supply voltages that are not provided by the ATX supply. In particular, the Raw chip requires a core supply of 1.8 V and an I/O supply of 1.5 V. The FPGAs also require 1.5 V for both their cores and for the I/Os that connect to Raw. Each of these four demands is met with a separate on-board switching DC/DC converter. Each converter is accompanied by a current monitoring circuit that allows the power drawn from each converter to be measured separately. This information was used to characterize the power consumption of the Raw processor under a variety of different conditions.

In addition to the 1.8 V and 1.5 V supplies, the HSTL signaling used for Raw's high-speed I/Os requires a 0.75 V termination and reference voltage. Because the HSTL I/Os are parallel-terminated to a voltage that is between the two rails, the termination voltage supply circuit must be capable of both sourcing and sinking current. In addition, the large number of HSTL I/O's in the system (over 1000) requires the supply to handle currents (in both directions) as high as 11 amps. Since HSTL I/Os use a very low voltage swing, the termination and reference voltage must also be extremely stable. Therefore a custom DC/DC converter circuit was designed by ISI and integrated onto the system board.

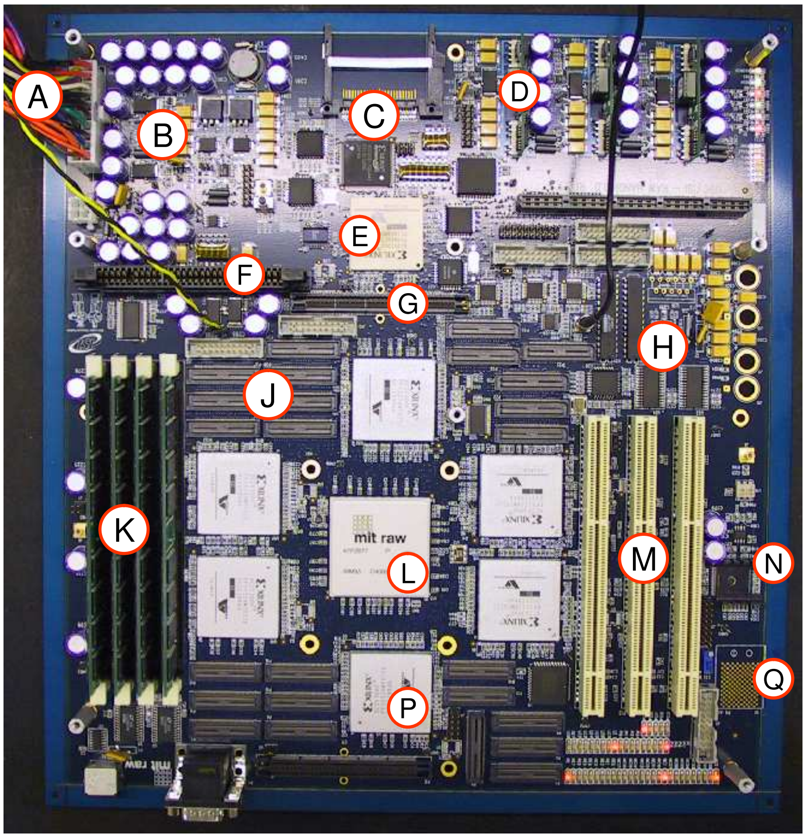
Debugging Support

Since the Single-chip system was the first system to use the prototype Raw chip, considerable care was taken to ensure that it had adequate debugging features. These features were used for testing the Raw processor and debugging both the board itself (including FPGA firmware) and the software that was being run on Raw. Several of these features have already been mentioned: programmable clocks, the USB host connection, and the implementation of the chipset in programmable logic. Several others are presented here.

Probably the most useful of all the debugging features added to the system board are the logic-analyzer headers on key Raw ports. All of the ports on the east and west sides of the Raw chip (connected to the MC0, MC1, PI and PCI FPGAs) can be probed using a high-speed logic analyzer. This permits observation of all messages going to and from DRAM, the PCI bus, the integrated I/O devices and the USB host connection. These headers consume considerable real-estate and greatly complicated routing but were invaluable for debugging both firmware and software. In addition to the headers on the Raw ports, there are one or two headers and several LEDs connected to unused pins on each of the FPGAs. These are very helpful for exporting debugging information from the firmware in the FPGAs.

When incorrect data is observed in the system, it is sometimes difficult to tell whether it is caused by errors in software/firmware or physical defects in the board. Because Raw and the FPGAs use high-density flip-chip packages, it is usually impossible to observe or probe a suspect pin on those chips. Therefore, all of the major chips in the system are connected together in JTAG boundary-scan chains. These help detect and isolate manufacturing defects that might otherwise be mistaken for programming bugs.

Instead of using the programmable clock generators, the Raw and memory clocks can be directly driven from an external clock source. This allows each of these components to be driven at virtually any clock speed from 0 to over 400 MHz. The Raw chip provides an



A) Power supply wires
B) 0.75 V DC/DC converter
C) SystemACE chip and CompactFlash slot
D) 1.5 V DC/DC converter
E) CONF FPGA
F) 1.8 V VRM slot
G) 190-pin expansion connector
H) Audio A/D and D/A converters
J) High-density logic-analyzer headers
K) 512 MB SDRAM DIMMs
L) Raw microprocessor
M) 64-bit PCI slots
N) 100 MHz A/D converter
P) Chipset FPGA (RE)
Q) Daughtercard connector (not installed)

Figure A-6: Photograph of the Raw Single-chip system board. The board uses an Extended ATX form factor and fits in a standard E-ATX PC chassis. Note that the daughtercard connector (Q) is not installed in this picture so the USB interface card is not shown.

external interface to its PLL configuration bits allowing for a wide variety of input clock frequencies and internal multiplication factors. Raw also provides a clock output pin to verify that the resulting internal clock is behaving as expected.

A.2.3 Host Software

Until it is mature enough to become a stand-alone system, the Raw Single-chip system is dependent on its connection to a host computer through the USB daughter-card. The host computer is responsible for resetting the Raw system, loading a program onto the Raw chip, providing the program with input, and collecting the program's output. Several pieces of software are required on the host to make all of this happen: a Linux USB driver, the Raw simulator (*BTL*), an interface layer between the driver and BTL, and a virtual device that runs in BTL called the *Host Interface*. The USB driver is simply responsible for sending and receiving raw data over the USB link. Each of the other components is discussed below.

The cycle-accurate simulator of the Raw chip is called *BTL*. In fact, BTL simulates more than just the chip itself. It must also provide external stimuli and a model for external DRAM in order to simulate the complete behavior of the chip in a real-world environment. In essence, BTL simulates the operation of an entire Raw-based system. To build these systems, BTL provides an extension language (called *bC*) that is used to write virtual devices. These devices can then be connected to the I/O ports on the simulated Raw chip to communicate with the tiles inside. Virtual devices can be written that do almost anything imaginable including interfacing with actual physical devices on the host machine. Some examples of virtual devices that have been created are: serial ROM (streams data into Raw), data logger (writes data from I/O port to disk file), DRAM controller and memory, network message loopback, PCI bus controller, interface to video capture card, and the Host Interface.

Once a machine model has been created, BTL provides the ability to substitute different models for the Raw chip. The default is a C++ model that provides fast, cycle-accurate results and supports powerful profiling and debugging tools. An alternative model is an RTL simulation of the Raw netlist using Synopsys VCS. BTL is connected to the VCS simulator using the Verilog PLI interface. Any values sent to the I/O ports by virtual devices are passed to VCS where they become external stimuli in the RTL simulation. Similarly, outputs from the RTL simulation are passed back to BTL which in turn passes them to the correct virtual device. (This mode of operation was the primary method used to verify the operation of the Raw netlist during development.) Once the Raw chip was finished, the natural desire was to swap out the RTL model for the actual chip. This would allow testing and software development using all of the regression tests and infrastructure created during chip development.

To accomplish this, a new piece of software was written that takes the place of the Raw chip in the BTL simulation and passes data to and from the actual Raw chip in a Single-chip system. It does this by prepending a header onto each word to be sent and then passing the result to the USB driver. When the packet arrives on the Single-chip system, firmware in the the PI and PCI FPGAs examines the header to determine for which I/O port and network (GDN, MDN or Static1) the data is intended. When the packet arrives at the correct port, the header is striped away and the data injected into the Raw chip. Similarly, when data comes out of the Raw chip, the FPGA firmware applies the header and sends it to the host where it is passed to the virtual device that is connected to the corresponding I/O port. In this way, the firmware, USB link, and BTL interface software form a bridge that connects the physical Raw chip to the simulated virtual devices that are running on the host computer. In the current implementation, only the I/O ports on the west side of the Raw chip are forwarded to the host computer. Therefore, messages sent to DRAM or the expansion FPGAs go to the physical devices on the system board.

One of the most important virtual devices is the Host Interface. This device provides basic operating system services by relaying system calls to the operating system of the computer running the simulation. When using the Single-chip system, this computer is the host machine. Until a complete operating system is written for Raw, programs must rely on the underlying host operating system for things like file I/O and display of output. C programs on Raw use the newlib C library with a port of libgloss to provide system calls. The Raw libgloss routines package up the system call arguments and send them to the Host Interface to execute the actual system call. Depending on the specific call (*e.g.*, `_read` or `_write`), the Host Interface may need to use DMA messages to transfer data to or from the Raw DRAMs. When the system call is complete it sends a message back to the Raw core with any return arguments. The libgloss routine receives the message and execution can continue. Because of the many steps involved and a quirk in the Linux USB layer on the host machine, the latency of system calls on the Single-chip system can be highly variable and unrealistically long. Therefore the Single-chip system is excellent for running large programs quickly (versus the simulator) but is less than ideal for performance studies that require accurate timing.

A.3 Fabric System

A.3.1 Design Goals and Overview

The Raw Fabric System [80] is designed to explore the scalability of the Raw architecture. It takes advantage of the fact that Raw chips can be gluelessly tiled together to simulate a Raw chip of the future. When the I/O ports of two Raw chips are connected together, they behave as if they were a single large Raw chip with twice as many tiles. There are only two differences that occur at the boundaries between chips: the second static network is not connected (see Section A.1.5) and there is an additional three cycles of latency on

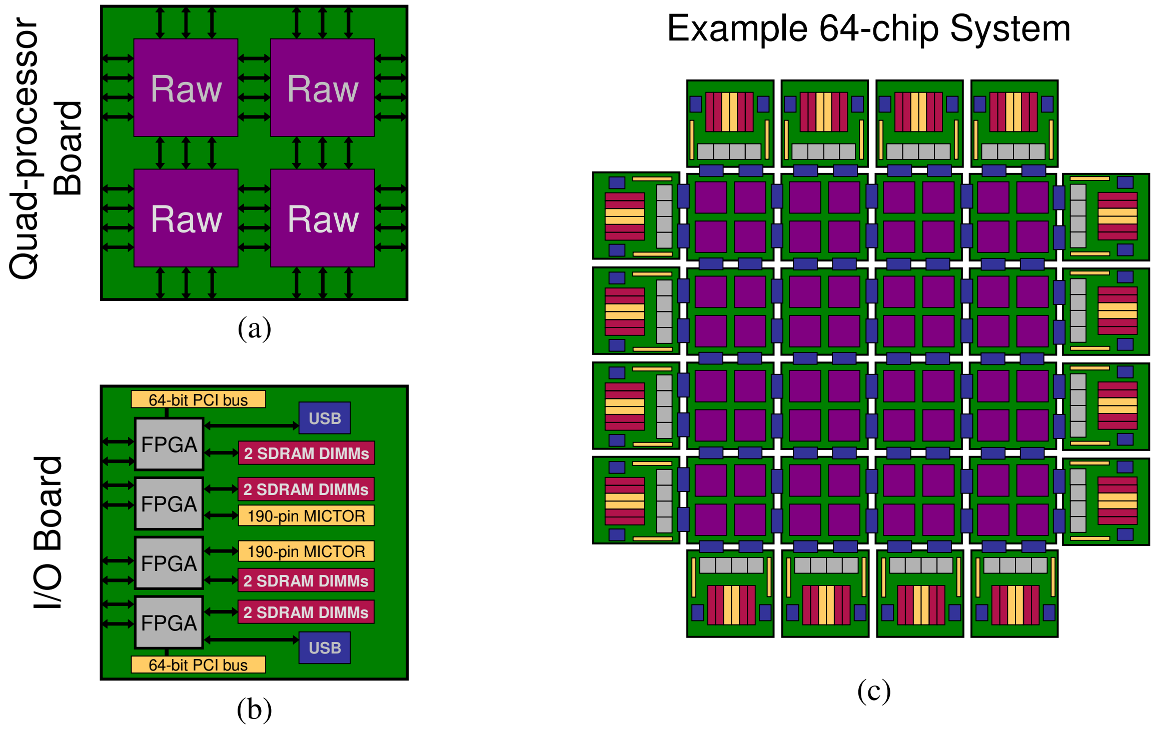


Figure A-7: Overview of the Raw Fabric system. The Quad board (a) provides the computational resources with four Raw processors. The I/O board (b) contains DRAM and I/O interfaces. A Fabric system (c) is composed of one or more Quad boards in a rectangular array and some number of I/O boards on the periphery.

the other networks due to the delay of the I/O drivers and receivers. Up to 64 chips can be connected together in an 8×8 array, simulating a 1024-core Raw chip that will be buildable in the 22 nm process generation.

However, a single PCB containing 64 Raw chips would probably have to be at least 36 inches on each side, making it extremely difficult to manufacture. Therefore a modular system of two smaller boards was designed. The first board contains a 2×2 array of Raw chips and is referred to as the *Quad-processor board* (Figure A-7(a)). The second board, containing four large FPGAs, eight SDRAM DIMMs, and a small assortment of I/O interfaces, is referred to as the *I/O board* (Figure A-7(b)). Each board also contains several high-density MICTOR connectors that allow it to be connected up to other boards. These two types of boards can be assembled in a variety of configurations to create systems with 4 to 64 Raw processors and as much I/O and DRAM bandwidth as needed (Figure A-7(c)).

The challenges of constructing a system in this way include providing adequate bandwidth between boards and coordinating the boards so that they behave as a cohesive unit.

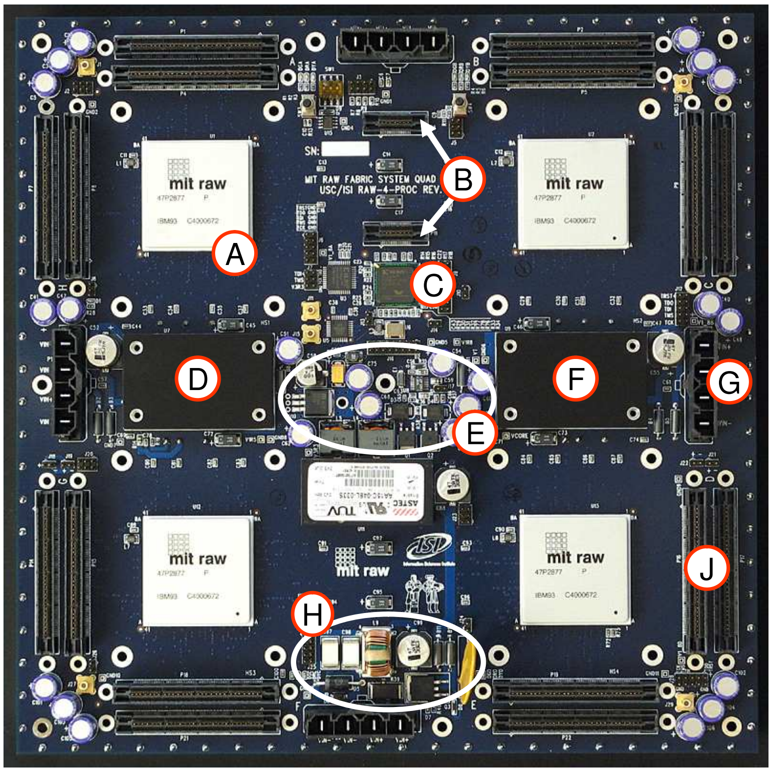
To provide the necessary bandwidth, each board-to-board junction consists of approximately 600 high-speed connections. These connections use high-density MICTOR connectors on the boards, joined together by impedance-matched ribbon cables with individually shielded conductors. To operate as a single unit, the entire array of Raw chips shares a single clock. This clock is distributed from board to board in a tree-like pattern and is deskewed by a PLL at each board. A global reset signal is similarly distributed to all boards. Finally, a maximum-sized system has the potential to draw about 3 kW of power in the worst case. Distributing this power at the low-voltages that the chips require would result in massive currents and unacceptable voltage drops. Therefore, power is distributed at 48 V and converted to the lower voltages locally on each board. The power supply is connected to the system at the edge and power is distributed across the array of boards in a mesh pattern using fat traces on the boards and heavy gauge wire between boards.

A.3.2 Quad-Processor Board

The Quad-processor board (*Quad* board, for short), provides the computational resources for a Fabric system. It is relatively simple compared the Single-chip system board and consists of only: four Raw processors, board-to-board connectors, DC/DC converters, and a very small FPGA. It is not capable of functioning as a complete computer by itself but must be paired with at least one I/O board to provide memory and I/O resources. Multiple Quad boards can be connected together in an array to create larger computational fabrics. The largest fabric allowed is a 4×4 array of Quad boards. Any rectangular subset of this array is also allowed as long as the number of boards on each side of the rectangle is a power of two. Therefore, Fabric systems can be built containing 4, 8, 16, 32 or 64 processors in several different shapes.

Raw Processors

The four Raw processors on the Quad board are connected together in a 2×2 array. Raw processors are designed so that the I/O ports on the east side of one chip can be directly connected to the I/O ports on the west side of neighboring chip. The same is true of the north and south I/O ports. In this way, the interconnection pattern of the chips mirrors the pattern of the on-chip networks within a chip. Also, similar to the situation on the chip, when the chip-to-chip connections reach the edge of the board, they are wired to connectors



A) Raw processor

B) Clock loopback connectors

C) Clock/reset FPGA

D) Eighth-brick 1.5V DC/DC converter

E) Custom 0.75V DC/DC converter

F) Eighth-brick 1.8V DC/DC converter

G) 48V power connector

H) 48V supply-isolation circuit

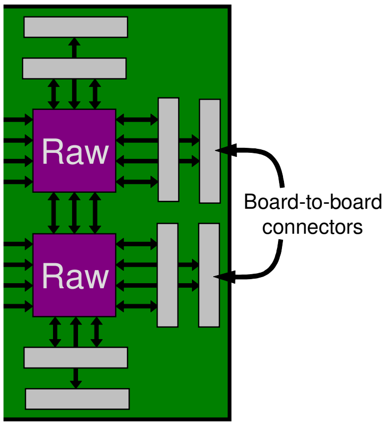
J) Board-to-board connectors (152-pin MICTOR)

Figure A-8: Photograph of the Raw Fabric system Quad-processor board. The board is 11 inches by 11 inches. The arrangement of signal and power connectors is the same on all sides, allowing these boards to be connected together in an array. To enable clock deskewing, a loopback cable of the same length as the board-to-board cables must be connected between the two connectors labeled B.

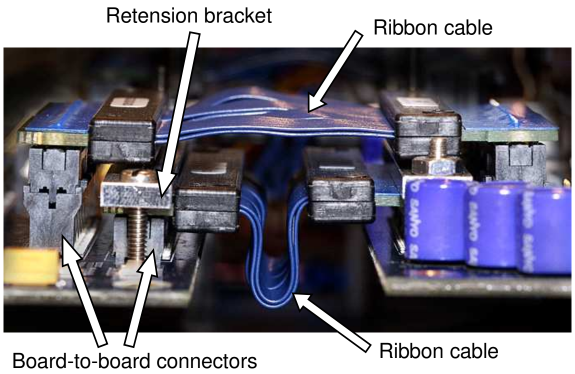
that can be used to bridge to the next board.

Board-to-Board Connections

Each east-west chip-to-chip connection consists of 296 wires. Since there are two such connections on each side of the board, connectors supporting almost 600 signals are required. Single connectors with 600 pins are difficult to find so four 152-pin connectors were used instead. These are the same style of MICTOR connectors used for the expansion connectors on the Single-chip system. Because the edge of the board is not long enough to fit all four connectors side-by-side, they are arranged in two rows (Figure A-9(a)). When two boards are connected together, the outer two connectors are joined with one ribbon cable and the inner connectors with another as shown in Figure A-9(b). The inner connectors are taller so that the cable connecting them will not physically interfere with the other cable.



(a)



(b)

Figure A-9: This figure shows the connectors and ribbon cables that join neighboring boards. Two rows of connectors (a) must be used to provide the 600 signals on each side of the board. The connectors have different heights (b) to accommodate two ribbon cables. Steel retention brackets ensure reliable matings of the connectors and ribbon cables.

This combination of connectors and cables has several advantages over more rigid options like backplane connectors. First, it allows any board in a large system to be connected or disconnected without disturbing the other boards. The ribbon cables only need to be detached from the board in question. If the boards were directly mated to each other, extensive disassembly of the system would be required to remove an inner board. Second, a directly-mated system would require enormous mating/demating forces for large system configurations. Boards would need to be assembled into strips and then the strips would be connected to form a mesh. When connecting one strip to another, 2400 pins would need to be mated simultaneously. Using cables, each 152-pin cable is dealt with individually. Third, a flexible board-to-board connection is more tolerant of manufacturing variations. Using direct mating or any other form of rigid interconnection requires all connectors in the system to be precisely aligned with each other. Typical PCB manufacturing methods result in small variations in component placement that make this extremely difficult to achieve. Flexible interconnects are able to change shape to accommodate small misalignments. Going one step further, the use of flexible interconnects allows for alternative system shapes. For example, rather than mounting all boards in a single plane, the array of boards could be wrapped around the inside or outside of a hexagonal cylinder. If the panels of the cylinder were connected by hinges, it could be opened up to allow access for system maintenance or repair. Flexible interconnects allow this to occur without any disassembly.

The disadvantage of connecting boards with multiple smaller cables is that it multiplies the number of connectors and therefore the potential for improper matings. With over 300 high-density connectors in a 64-chip system, loose connections can easily create a debugging nightmare. To help ensure that the MICTOR connectors are fully seated and remain that way, steel retention brackets are used on each connector. These brackets sandwich the connectors from both sides (one on the bottom of the board and one on top of the connectors) and are pulled together by screws. This holds the connectors together very effectively but also tends to warp the board slightly. This warping can result in broken solder joints where the connectors are attached to the board. To prevent this, metal board stiffeners are used along the full length of all four sides of the board. Additional warping of the board can be caused by the heatsinks mounted over the Raw chips. For this reason, the entire board must be attached to a flat rigid substrate using stand-offs distributed across the board.

Power Distribution and Conversion

As mentioned previously, power is distributed throughout the system at 48 V. This power enters and exits each board through power connectors located in the middle of each side of the board. Short jumper cables are used to connect one board to the next. The four connectors are wired together on the board using heavy traces. Switching DC/DC converters on the board tap into the 48 V supply and provide the 0.75 V, 1.5 V, 1.8 V and 3 V required by the various chips. An isolation circuit between the external supply and the converters prevents supply noise or improperly connected wires from damaging the internal components on the board. The 1.5 V and 1.8 V converters are off-the-shelf parts utilizing the standard *quarter-brick* form-factor. These parts are large and therefore difficult to fit into the layout. However, since they only have pins along the two short edges, they could be placed so that they straddle the large buses of wires running between Raw chips. This space would otherwise have been wasted since the density of wires in the buses makes it impossible to place other components there.

Clock and Reset Distribution

The final component of the Quad board is a small FPGA located near the center of the board. This FPGA is responsible for programming the PLLs in the Raw chips (on startup) and distributing the global clock and reset signals. The Quad board is designed so that

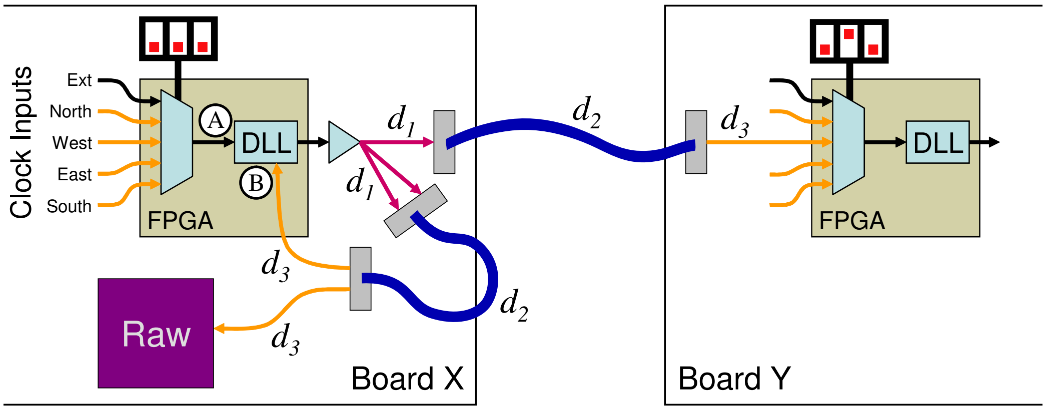


Figure A-10: Clock distribution and deskewing pathways. An input clock is selected on board X using the three DIP switches. This signal is then distributed locally and to the neighboring boards. Trace and cable lengths are matched so that all copies of the clock experience the same delay ($d_1 + d_2 + d_3$). The DLL on board X compensates for this delay (by shifting its output clock) so that there is zero skew between the clocks at A and B. Since the delays are matched, all copies of the clock are now synchronized.

it can receive clock and reset signals from any of its neighboring boards or from a direct external input. A set of DIP switches are used to specify which direction should be treated as the input. The signal coming from the input direction is then broadcast to the other directions so that the signal will be propagated to the other boards in the system. By setting switches on all of the Quad boards appropriately, the clock and reset signals can originate anywhere (using the external input) and be distributed to all the other boards in various tree-like patterns. The reset signal would typically originate on an I/O board that is connected to a host computer or other user interface device. The clock signal would typically originate on a Quad board near the physical center of the system to minimize the depth of the distribution tree.

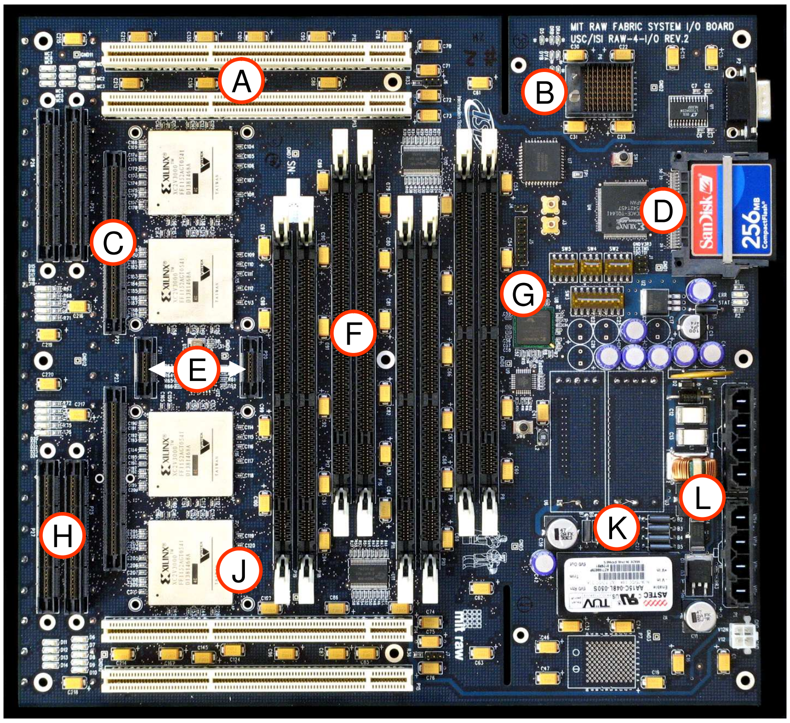
It is not crucial that the reset signal arrive at all boards simultaneously since it is assumed that it will be asserted for several cycles. Therefore it is simply passed from one board to the next as it is received. However, the clock signal must be precisely coordinated across the entire system since all I/O on Raw is synchronous. The relative clock skew between adjacent boards will directly affect the maximum clock frequency of the system. Therefore, the clock is deskewed using a DLL in the FPGA each time it is passed to the next board. The DLL takes a clock as input and produces a phase-shifted copy of the clock for distribution to the Raw chips and neighboring boards. It deskews the clock by adjusting the phase shift so that a clock edge arrives at the distribution endpoints at exactly the same time as an edge arrives at the input to the DLL. To do this, it requires that all distribution paths have the same delay (*i.e.*, the same wire lengths) and that one of the paths loops back to the DLL as a feedback input (Figure A-10). The difficulty with this requirement is that some of the distribution endpoints are on neighboring boards, across ribbon cables of unknown length. To compensate for this, two things are required. First, all ribbon cables in the system must be the same length. Second, a separate *loopback* ribbon cable (of the same length as the other cables) must be installed with both ends on the same board. Now, the copies of the clock destined for Raw chips and the feedback input of the DLL must be routed over the loopback cable. As long as the rest of the clock traces on the board have matched lengths, this will ensure that the distribution paths within a board have the same delay as those going to neighboring boards. Since this is repeated on every board, all Raw chips in the system will receive clock edges at the same time.

A.3.3 I/O Board

As the name implies, the *I/O board* provides memory and I/O resources for a Raw Fabric system. It essentially provides the same functions as the FPGA chipset and integrated devices found in the Single-chip system except that it focuses on general-purpose I/O and omits the specialized devices. I/O boards may be connected anywhere on the periphery of an array of Quad boards. At least one I/O board is required in every Fabric system. Additional I/O boards can be added (until the perimeter is full) to provide additional memory or I/O bandwidth.

Major Components

Whenever possible, the I/O board uses the same components as the Single-chip system board. Each I/O board contains four large Virtex-II FPGAs to interface with the Raw chips on a Quad board and implement device controllers. The center two FPGAs are designated EXP0 and EXP1 while the outer two are called PCI0 and PCI1. As with the MC FPGAs on the Single-chip system, each FPGA is connected to two DIMMs on separate buses. In addition to memory, each FPGA is connected to some form of I/O interface. The EXP FPGAs each have a single 190-pin MICTOR connector like those found on the Single-chip system. The PCI FPGAs each have a two-slot 64-bit PCI bus and a Teradyne

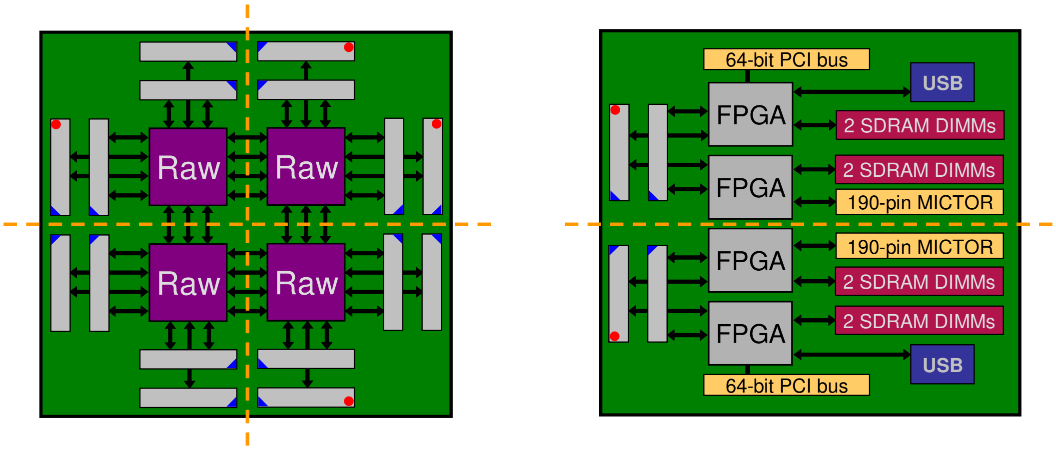


A) 64-bit PCI bus
B) Daughtercard connector
C) Expansion connector
D) SystemACE chip & CompactFlash card
E) Clock loopback connectors
F) SDRAM DIMM sockets
G) Clock/reset FPGA
H) Board-to-board connectors
J) Chipset FPGA (PCI0)
K) DC/DC converters (some on back)
L) 48V power supply connectors and isolation circuit

Figure A-11: Photograph of the Fabric I/O board. The board is 11 inches high and 12 inches wide. The connectors on the left side (H) allow it to be connected to any side of a Quad board. The four FPGAs (*e.g.*, J) interface to the Raw processors on the Quad board and implement controllers for the PCI buses (A), daughtercards (B), and DRAM (F).

daughtercard connector for a USB interface card. This allows the I/O board to use the same firmware as the Single-chip system (with only minor changes) for the memory, PCI and USB controllers.

Many of the other components on the I/O board are reused from other boards as well. In addition to the large FPGAs, a single smaller FPGA is used for clock and reset distribution just like on the Quad board. All of the FPGAs on the I/O board are programmed using the same Xilinx SystemACE solution that is used on the Single-chip system. Of course, the I/O board also uses the same ribbon-cable retention brackets and board stiffeners found on the Quad board. The Single-chip and Fabric systems can also share the same host software because they use the same USB interface card and controller firmware. Since the Raw chip was designed to be tiled together into large systems right from the beginning, the BTL simulator natively supports machine configurations that match the Fabric system. Only minor changes were required to the interface layer between BTL and the USB driver when the new system was brought on-line.



(a) (b)

Figure A-12: Connector symmetry required to use an I/O board on any side of a Quad board. Dashed yellow lines show axes of symmetry. Blue triangles indicate pin 1 position (enforced by keyed connectors). Red dots represent an asymmetrical signal such as reset. The mirror-image position of this signal must be empty on the Quad board (a) and connected as a second reset signal on the I/O board (b). The clock/reset FPGA on the I/O board can be reprogrammed to use the correct copy, depending on where it is connected.

Design Choices and Challenges

An alternative design for the Fabric system might have used two boards in place of the I/O board: one for memory and another for I/O. This would seem to be advantageous when, for example, building systems that require a lot of memory bandwidth but very little I/O. In reality, most of the components on the I/O board (*e.g.*, FPGAs, power supplies, board-to-board connectors) would be required on both types of smaller boards. Thus it was cheaper and easier to create a single board with all of the different types of connectors. If single-function boards are required and costs must be minimized, the I/O connectors or DIMM sockets can be omitted during assembly. Another advantage of a single board that incorporates both I/O and memory is the ability to share a single I/O port for both uses. This is particularly helpful on systems that require maximum memory bandwidth and therefore memory on all I/O ports in the system. With single-function boards, an entire memory board would need to be omitted to allow for a single low-speed I/O device. With dual-purpose boards, all memory banks can be populated and bandwidth will only be lost when I/O communication actually occurs.

The challenge with building only a single type of I/O board is that it must be designed so that it can be used on any side of an array of Quad boards. This essentially involves carefully selecting the arrangement and pinout of the board-to-board connectors on both the Quad board and I/O board. The problem is complicated by the fact that the Quad boards must connect to each other as well as to the I/O boards. Fortunately, the use of FPGAs on the I/O board helps enormously. By loading different firmware into I/O boards on different sides of the array, FPGA pins can be remapped to the correct pinout on each side. However, the clock and reset lines use special signaling and rely on other discrete components to function. These lines cannot simply be remapped in the FPGA. The key to creating a compatible arrangement is to maintain axial symmetry in each set of connectors as shown in Figure A-12. One half of the board must be a mirror image of the other half. This includes the orientation of the connectors (since they are keyed and can only be connected to cables in one way) as well as the arrangement of signals on those connectors. Sometimes this requires duplicating signals on the I/O board connectors and programming the FPGAs to use the correct copy depending on which side of the array the board is used.

A.4 Summary

This appendix provided an overview of the Raw microprocessor, Raw Single-chip system and Raw Fabric system. The Raw microprocessor is a tiled multicore chip containing 16 tiles with tightly-integrated on-chip communication networks. These networks are used for core-to-core communication as well as off-chip I/O and memory access. The processing core in each tile is deliberately kept simple to allow for the maximum amount of general-purpose resources on a chip. Of particular interest to this thesis is the use of an explicitly-managed instruction SRAM rather than a hardware instruction cache. Here is a list of the key features of Raw that are relevant to the Flexicache system:

- Each Raw tile operates independently and resembles the abstract processor model shown in Figure 2-1.
  - Eight-stage, single-issue, in-order pipeline with MIPS-style ISA. [§ A.1.2]
  - 32 KB explicitly-managed instruction memory (I-mem). [§ A.1.2]
  - Special instructions (`ilw` and `isw`) to read and write I-mem locations. [§ A.1.2]
  - Non-blocking DRAM access via MDN network messages. [§ A.1.5]
- Until the Raw systems are mature enough to become stand-alone, system calls are proxied to a host computer using MDN messages. [§ A.2.3]

The Single-chip and Fabric systems combine Raw chips with external DRAM and interfaces to I/O devices that allow Raw to realize its enormous computational potential. The Single-chip system is a workstation-class machine with one Raw processor, 2 GB of DRAM, a PCI bus, integrated I/O devices, and a variety of features for debugging and experimentation. The Fabric system is a larger machine that is scalable from 4 to 64 Raw chips. It is designed to demonstrate the scalability of the Raw architecture and provide supercomputer performance. The Flexicache implementation described in Chapter 4 as well as the experimental evaluations in Chapter 7 are based on these systems.

Appendix B

Rewriter and Runtime Flowcharts

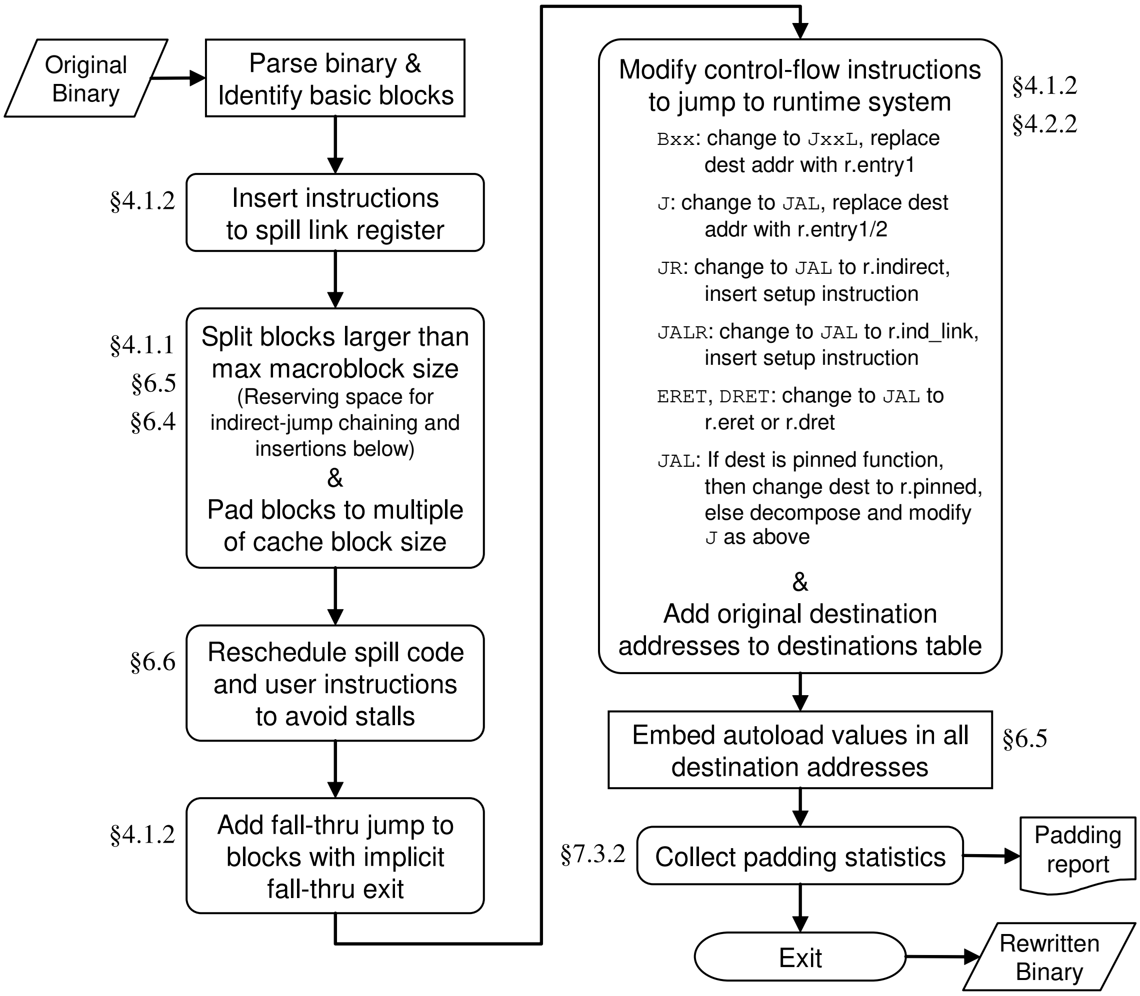


Figure B-1: High-level operation of Flexicache preprocessor. Rounded-corner boxes represent tasks that iterate over all basic/cache blocks in the program. Refer to indicated sections for more information.

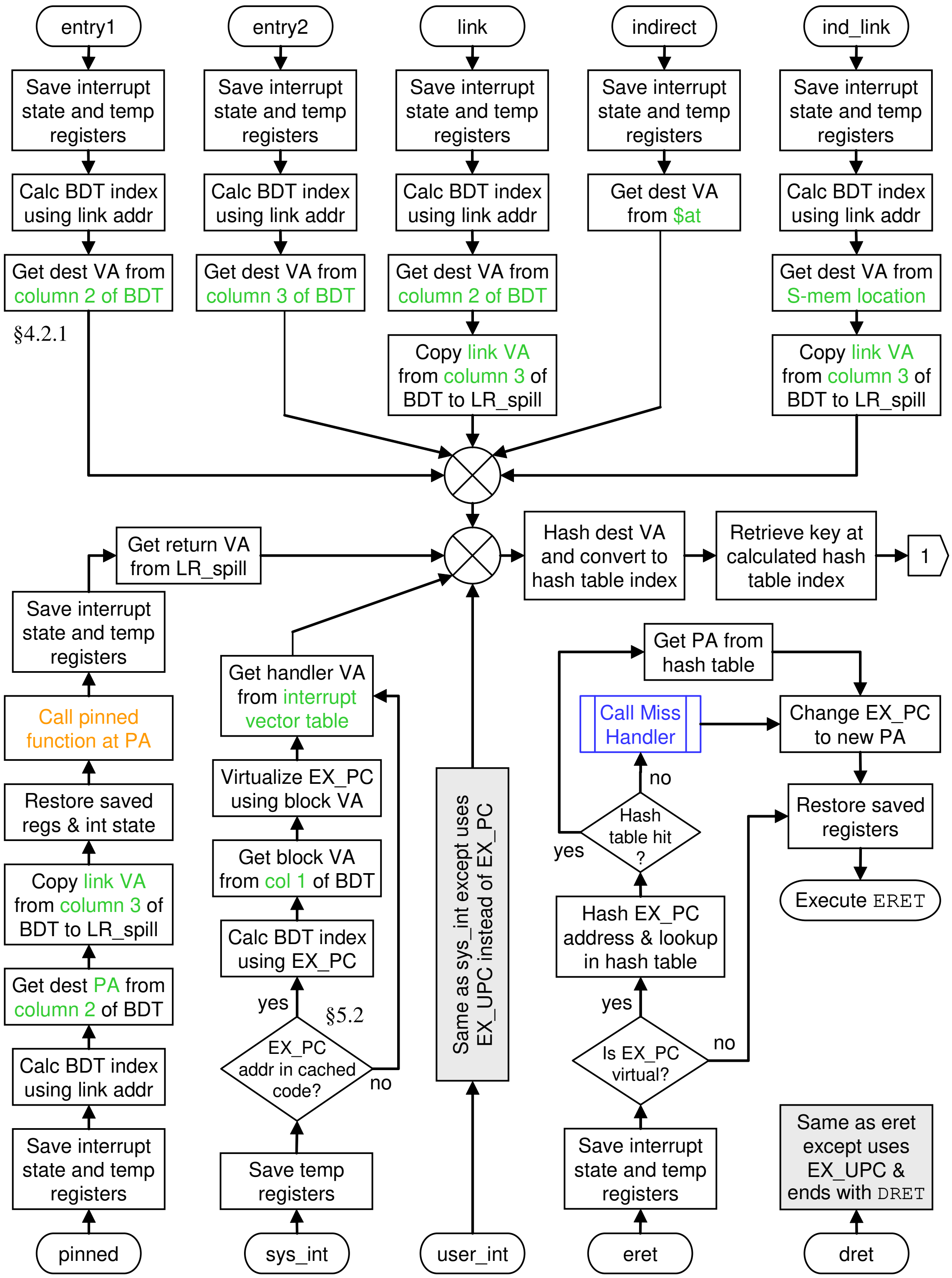


Figure B-2: Operation of Flexicache runtime system entry point routines. Colors highlight key differences between entry points. Refer to indicated sections for more information. Runtime operation continues after hash-table lookup on next page. See next page for definition of abbreviations.

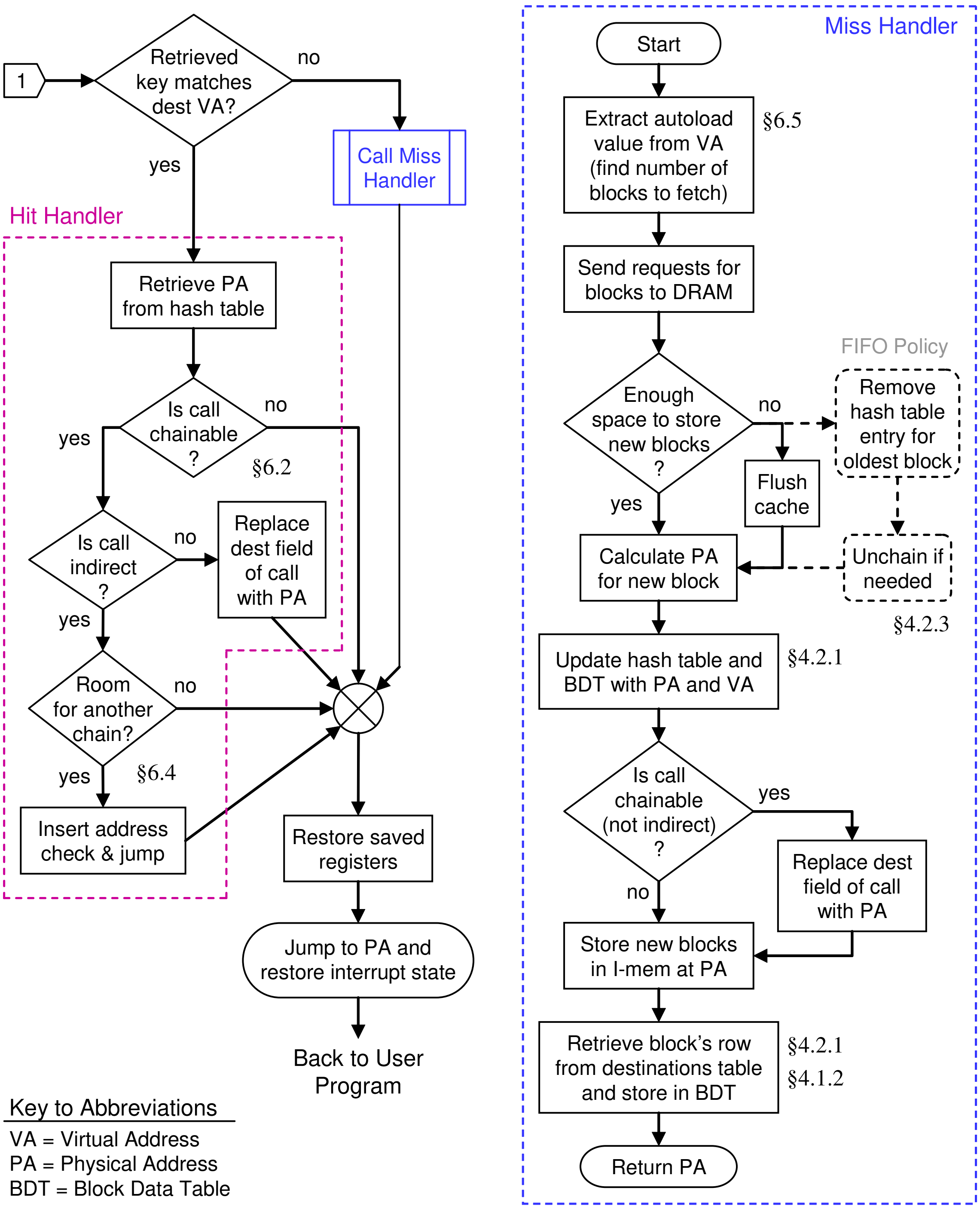


Figure B-3: Operation of Flexicache runtime system hit and miss handlers. The miss handler is shown using the Flush replacement policy with the FIFO policy as an alternate path. Refer to indicated sections for more information.

Bibliography

[1] *PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors.* International Business Machines Corporation, Hopewell Junction, NY, 2000.

[2] Matthew Adiletta, Mark Rosenbluth, Debra Bernstein, Gilbert Wolrich, and Hugh Wilkinson. The next generation of Intel IXP network processors. *Intel Technology Journal*, 6(3):6–18, August 2002.

[3] Ali-Reza Adl-Tabatabai, Michał Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 280–290, June 1998.

[4] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *ISCA '00: Proceedings of the 27th annual International Symposium on Computer Architecture*, pages 248–259, 2000.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, Reading, MA, March 1988.

[6] Federico Angiolini, Francesco Menichelli, Alberto Ferrero, Luca Benini, and Mauro Olivieri. A post-compiler approach to scratchpad mapping of code. In *CASES '04: Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 259–267, Sep 2004.

[7] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 1–12, June 2000.

[8] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *CODES '02: Proceedings of the tenth International Symposium on Hardware/Software Codesign*, pages 73–78, May 2002.

[9] Swagato Basumallick and Kelvin D. Nilsen. Cache issues in real-time systems. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, Jun 1994.

[10] Bob Bentley. High level validation of next-generation microprocessors. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop*, pages 31–35, Oct 2002.

[11] Vanu Bose, Mike Ismert, Matt Welborn, and John Guttag. Virtual radios. *IEEE Journal on Selected Areas in Communications*, 17(4), April 1999.

[12] R. A. Brooker. An attempt to simplify coding for the Manchester electronic computer. *British Journal of Applied Physics*, 6(9):307–311, 1955.

[13] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA '00: Proceedings of the 27th annual International Symposium on Computer Architecture*, pages 83–94, 2000.

[14] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for Windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, Austin, Texas, December 2001.

[15] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 2004.

[16] Jose V. Busquets-Mataix, Juan J. Serrano, Rafael Ors, Pedro Gil, and Andy Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *RTAS '96: Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, pages 204–212, 1996.

[17] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M. Gillies. Mojo: A dynamic optimization system. In *Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 81–90, 2000.

[18] David R. Cheriton, Gert A. Slavenburg, and Patrick D. Boyle. Software-controlled caches in the VMP multiprocessor. In *ISCA '86: Proceedings of the 13th annual International Symposium on Computer Architecture*, pages 366–374, 1986.

[19] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and Modeling of Computer Systems*, pages 128–137, 1994.

[20] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report SMLI 93-12, UWCSE 93-06-06, Sun Microsystems Laboratories, Inc. and the University of Washington, 1993.

[21] Standard Performance Evaluation Corporation, 2002. `http://www.spec.org/`.

[22] Peter J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, 1970.

[23] Peter J. Denning. Before memory was virtual. In Robert L. Glass, editor, *In the Beginning: Recollections of Software Pioneeers*, pages 250–271. IEEE Press, 1997.

[24] Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, and Joseph A. Fisher. DELI: A new run-time control point. In *MICRO-35: Proceedings of the 35th annual International Symposium on Microarchitecture*, pages 257–268, Nov 2002.

[25] Angel Dominguez, Sumesh Udayakumaran, and Rajeev Barua. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing*, 1(4), 2005.

[26] Kemal Ebcioglu and Erik R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *ISCA '97: Proceedings of the 24th annual International Symposium on Computer Architecture*, pages 26–37, June 1997.

[27] Kemal Ebcioglu, Erik R. Altman, Michael Gschwind, and Sumedh W. Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on Computers*, 50(6):529–548, 2001.

[28] Alexandre E. Eichenberger, John Kevin OBrien, Kathryn M. OBrien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, Michael K. Gschwind, Roch Archambault, Yaoqing Gao, and Roland Koo. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture. *IBM Systems Journal*, 45(1):59–84, January 2006.

[29] Dawson R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 160–170, May 1996.

[30] Joshua B. Fryman, Chad M. Huneycutt, and Kenneth M. Mackenzie. Investigating a SoftCache via dynamic rewriting. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2001.

[31] Joshua B. Fryman, Hsien-Hsin S. Lee, Chad M. Huneycutt, Naila F. Farooqui, Kenneth M. Mackenzie, and David E. Schimmel. SoftCache: A technique for power and area reduction in embedded systems. Technical Report GIT-CERCS-03-06, Georgia Institute of Technology CERCS, 2003.

[32] Simcha Gochman, Avi Mendelson, Alon Naveh, and Efraim Rotem. Introduction to the Intel Core Duo processor architecture. *Intel Technology Journal*, 10(2):90–97, May 2006.

[33] Simcha Gochman, Ronny Ronen, Ittai Anati, Ariel Berkovits, Tsvika Kurts, Alon Naveh, Ali Saeed, Zeev Sperber, and Robert C. Valentine. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 7(2):21–36, May 2003.

[34] Ed Grochowski and Murali Annavaram. Energy per instruction trends in Intel microprocessors. *Technology@Intel Magazine*, March 2006.

[35] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10–24, March-April 2006.

[36] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mary Jane Irwin, N. Vijaykrishnan, Mahmut Kandemir, Tao Li, and Lizy Kurian John. Using complete machine simulation for software power estimation: The SoftWatt approach. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 141–150, 2002.

[37] Erik G. Hallnor and Steven K. Reinhardt. A fully associative software-managed cache design. In *ISCA '00: Proceedings of the 27th annual International Symposium on Computer Architecture*, pages 107–116, 2000.

[38] Kim Hazelwood and James E. Smith. Exploring code cache eviction granularities in dynamic optimization systems. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, page 89, 2004.

[39] Kim Hazelwood and Michael D. Smith. Code cache management schemes for dynamic optimizers. In *Sixth Annual Workshop on Interaction between Compilers and Computer Architectures*, pages 102–110, Boston, MA, February 2002.

[40] Kim Hazelwood and Michael D. Smith. Managing bounded code caches in dynamic binary optimization systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 3(3):263–294, 2006.

[41] Joe Heinrich. *MIPS R4000 Microprocessor User's Manual*. MIPS Technologies, Mountain View, CA, 1994.

[42] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Francisco, CA, third edition, 2003.

[43] Patrick Hicks, Matthew Walnock, and Robert Michael Owens. Analysis of power consumption in memory hierarchies. In *ISLPED '97: Proceedings of the 1997 International Symposium on Low Power Electronics and Design*, pages 239–242, 1997.

[44] Ron Ho, Kenneth W. Mai, and Mark A. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, April 2001.

[45] Chad M. Huneycutt, Joshua B. Fryman, and Kenneth M. Mackenzie. Software caching using dynamic binary rewriting for embedded devices. In *International Conference on Parallel Processing*, pages 621–630, Aug 2002.

[46] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7(1-2):229–248, 1993.

[47] Bruce L. Jacob and Trevor N. Mudge. Software-managed address translation. In *HPCA '97: Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, pages 156–167, Feb 1997.

[48] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[49] Ho-Seop Kim and James E. Smith. Hardware support for control transfers in code caches. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 253, 2003.

[50] Ilhyun Kim and Mikko H. Lipasti. Half-price architecture. In *ISCA '03: Proceedings of the 30th annual International Symposium on Computer Architecture*, pages 28–38, June 2000.

[51] Jason Sungtae Kim, Michael Bedford Taylor, Jason Miller, and David Wentzlaff. Energy characterization of a tiled architecture processor with on-chip networks. In *ISLPED '03: Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, pages 424–427, August 2003.

[52] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *USENIX Security Symposium*, San Francisco, August 2002.

[53] AJ KleinOsowski and David J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, Jun 2002.

[54] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanovic. The vector-thread architecture. In *ISCA '04: Proceedings of the 31st annual International Symposium on Computer Architecture*, pages 52–63, June 2004.

[55] Chandra J. Krintz, David Grove, Vivek Sarkar, and Brad Calder. Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31(8):717–738, 2001.

[56] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO-30: Proceedings of the 30th annual International Symposium on Microarchitecture*, pages 330–335, 1997.

[57] Minsuk Lee, Sang Lyul Min, Chang Yun Park, Young Hyun Bae, Heonshik Shin, and Chong-Sang Kim. A dual-mode instruction prefetch scheme for improved worst case and average case program execution times. In *IEEE Real-Time Systems Symposium*, pages 98–105, 1993.

[58] J. S. Liptay. Structural aspects of the System/360 Model 85, Part II: The cache. *IBM Systems Journal*, 7(1):15–21, 1968.

[59] Albert Ma, Michael Zhang, and Krste Asanovic. Way memoization to reduce fetch energy in instruction caches. In *Workshop on Complexity-Effective Design, 28th annual International Symposium on Computer Architecture (ISCA '01)*, June 2001.

[60] Philip Machanick, Pierre Salverda, and Lance Pompe. Hardware-software trade-offs in a direct Rambus implementation of the RAMpage memory hierarchy. In *ASPLOS '98: Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–114, 1998.

[61] Jason E. Miller. Software based instruction caching for the RAW architecture. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1999.

[62] Jason E. Miller and Anant Agarwal. Software-based instruction caching for embedded processors. In *ASPLOS '06: Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 293–302, October 2006.

[63] James Montanaro, Richard T. Witek, Krishna Anne, Andrew J. Black, Elizabeth M. Cooper, Daniel W. Dobberpuhl, Paul M. Donahue, Jim Eno, Gregory W. Hoeppner, David Kruckemyer, Thomas H. Lee, Peter C. M. Lin, Liam Madden, Daniel Murray, Mark H. Pearce, Sribalan Santhanam, Kathryn J. Snyder, Ray Stephany, and Stephen C. Thierauf. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *IEEE JSSC*, 31(11):1703–1714, November 1996.

[64] Csaba Andras Moritz, Matthew Frank, and Saman P. Amarasinghe. FlexCache: A framework for flexible compiler generated data caching. In *IMS '00: Revised Papers from the Second International Workshop on Intelligent Memory Systems*, pages 135–146, London, UK, 2001. Springer-Verlag.

[65] Csaba Andras Moritz, Matthew Frank, Walter Lee, and Saman Amarasinghe. Hot pages: Software caching for Raw microprocessors. Technical Report LCS-TM-599, Massachusetts Institute of Technology Lab for Computer Science, 1999.

[66] Henk Muller, David May, James Irwin, and Dan Page. Novel caches for predictable computing. Technical Report CSTR-98-011, Department of Computer Science, University of Bristol, Oct 1998.

[67] Peter Naur. The performance of a system for automatic segmentation of programs within an ALGOL compiler (GIER ALGOL). *Communications of the ACM*, 8(11):671–676, 1965.

[68] Kevin B. Normoyle, Michael A. Csoppenszky, Allan Tzeng, Timothy P. Johnson, Christopher D. Furman, and Jamshid Mostoufi. UltraSPARC-IIi: Expanding the boundaries of a system on a chip. *IEEE Micro*, 18(2):14–24, 1998.

[69] University of Southern California's Information Sciences Institute East. `http://www.east.isi.edu`.

[70] Subbarao Palacharla. *Complexity-Effective Superscalar Processors*. PhD thesis, University of Wisconsin–Madison, 1998.

[71] R. J. Pankhurst. Operating systems: Program overlay techniques. *Communications of the ACM*, 11(2):119–125, 1968.

[72] Peter Petrov and Alex Orailoglu. Energy frugal tags in reprogrammable I-caches for application-specific embedded processors. In *CODES '02: Proceedings of the tenth International Symposium on Hardware/Software Codesign*, pages 181–186, May 2002.

[73] James Ryan Psota. rMPI: An MPI-compliant message passing library for tiled architectures. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 2006.

[74] Rajiv A. Ravindran, Pracheeti D. Nagarkar, Ganesh S. Dasika, Eric D. Marsman, Robert M. Senger, Scott A. Mahlke, and Richard B. Brown. Compiler managed dynamic instruction placement in a low-power code cache. In *CGO '05: Proceedings of the International Symposium on Code Generation and Optimization*, pages 179–190, March 2005.

[75] Jesus Sanchez and Antonio Gonzalez. Modulo scheduling for a fully-distributed clustered VLIW architecture. In *MICRO-33: Proceedings of the 33rd annual International Symposium on Microarchitecture*, pages 124–133, December 2000.

[76] Premkishore Shivakumar and Norman P Jouppi. CACTI 3.0: An integrated cache timing, power and area model. Technical Report 2001/2, Compaq Western Research Laboratory, Dec 2001.

[77] Timothy J. Slegel, Robert M. Averill III, Mark A. Check, Bruce C. Giamei, Barry W. Krumm, Christopher A. Krygowski, Wen H. Li, John S. Liptay, John D. MacDougall, Thomas J. McPherson, Jennifer A. Navarro, Eric M. Schwarz, Kevin Shum, and Charles F. Webb. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, 1999.

[78] Thomas R. Spacek. A proposal to establish a pseudo virtual memory via writable overlays. *Communications of the ACM*, 15(6):421–426, 1972.

[79] Stefan Steinke, Lars Wehmeyer, Bo-Sik Lee, and Peter Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *DATE '02: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 409–417, Mar 2002.

[80] Albert G. Sun. Raw Fabric hardware implementation and characterization. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 2006.

[81] Weiyu Tang, Alexander V. Veidenbaum, Alexandru Nicolau, and Rajesh K. Gupta. Integrated I-cache way predictor and branch target buffer to reduce energy consumption. In *ISHPC '02: Proceedings of the 4th International Symposium on High Performance Computing*, pages 120–132, 2002.

[82] Michael Taylor. *Tiled Processors*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, February 2007.

[83] Michael Taylor. The Raw processor specification. Technical report, Massachusetts Institute of Technology, Continuously updated 2005. `http://cag.csail.mit.edu/raw/documents/RawSpec99.pdf`.

[84] Michael Bedford Taylor, Jason Kim, Jason Eric Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, Mar 2002.

[85] Michael Bedford Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Scalar operand networks: On-chip interconnect for ILP in partitioned architectures. In *HPCA '03: Proceedings of the 9th International Symposium on High Performance Computer Architecture*, pages 341–353, Feb 2003.

[86] Michael Bedford Taylor, Walter Lee, Saman P. Amarasinghe, and Anant Agarwal. Scalar operand networks. *IEEE Transactions on Parallel and Distributed Systems*, 16(2):145–162, February 2005.

[87] Michael Bedford Taylor, Walter Lee, Jason Eric Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *ISCA '04: Proceedings of the 31st annual International Symposium on Computer Architecture*, pages 2–13, June 2004.

[88] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *Transactions on Embedded Computing Systems*, 5(2):472–511, May 2006.

[89] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic overlay of scratch-pad memory for energy minimization. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/software Codesign and System Synthesis*, pages 104–109, 2004.

[90] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86–93, Sept 1997.

[91] Benjamin Philip Eugene Zaks Walker. A Raw processor interface to an 802.11b/g RF front end. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 2004.

[92] Eugene Weinstein, Kenneth Steele, Anant Agarwal, and James Glass. LOUD: A 1020-node modular microphone array and beamformer for intelligent computing spaces. Technical Report MIT-CSAIL-TR-2004-021, Massachusetts Institute of Technology CSAIL, April 2004.

[93] David Wentzlaff and Anant Agarwal. Constructing virtual architectures on a tiled processor. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 173–184, 2006.

[94] Maurice V. Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions on Electronic Computers*, EC-14(2):270–271, 1965.

[95] Steven J. E. Wilton and Norman P. Jouppi. CACTI: An enhanced cache access and cycle time model. *IEEE JSSC*, 31(5):677–688, May 1996.

[96] Emmett Witchel, Sam Larsen, C. Scott Ananian, and Krste Asanovic. Direct addressed caches for reduced power consumption. In *MICRO-34: Proceedings of the 34th annual International Symposium on Microarchitecture*, December 2001.

[97] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 68–79, May 1996.

[98] Se-Hyun Yang, Babak Falsafi, Michael D. Powell, and T. N. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 151–161, Feb 2002.

[99] Michael Zhang and Krste Asanovic. Highly associative caches for low-power processors. In *Kool Chips Workshop, 33rd International Symposium on Microarchitecture*, 2000.