Software-based Instruction Caching for Embedded Processors

Jason E Miller   Anant Agarwal

Computer Science and Artificial Intelligence Lab (CSAIL)
Massachusetts Institute of Technology
Cambridge, MA 02139
jasonm@alum.mit.edu, agarwal@mit.edu

Abstract

While hardware instruction caches are present in virtually all general-purpose and high-performance microprocessors today, many embedded processors use SRAM or scratchpad memories instead. These are simple array memory 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. Access times are also easier to predict with simple memories since there is no possibility of a "miss." On the other hand, they are more difficult for the programmer to use since they are not automatically managed.

In this paper, we present a software system that allows all or part of an SRAM or scratchpad memory to be automatically managed as a cache. This system provides the programming convenience of a cache for processors that lack dedicated caching hardware. It has been implemented for an actual processor and runs on real hardware. Our results show that a software-based instruction cache can be built that provides performance within 10% of a traditional hardware cache on many benchmarks while using a cheaper, simpler, SRAM memory. On these same benchmarks, energy consumption is up to 3% lower than it would be using a hardware cache.

Categories and Subject Descriptors B.3.2 [*Memory Structures*]: Design Styles—Cache memories; D.4.2 [*Operating Systems*]: Storage Management—Storage hierarchies, virtual memory; C.4 [*Performance of Systems*]: Design studies; C.3 [*Special-purpose and Application-based Systems*]: Real-time and embedded systems

General Terms Design, Experimentation, Measurement, Performance

Keywords Software caching, instruction cache, chaining

1. Introduction

As portable devices such as cellular phones, PDAs and MP3 players play a larger role in society, people expect increasingly sophisticated functionality from them. These devices are quickly becoming universal communication and media interfaces, incorporating a wide range of computationally intensive applications. The key to enabling these applications is higher-performance embedded processors. However, the ever-increasing gap between processor and DRAM speeds is a major obstacle to acheiving this performance. General-purpose processors mitigate long memory access latencies by introducing caches into the memory hierarchy. However, caches have not been broadly accepted in embedded systems for a variety of reasons including: design and manufacturing cost, power consumption and timing uncertainty.

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 in the cache. This makes programming processors with caches easy since the programmer does not need to explicitly manage the memory hierarchy. However, hardware caches 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 consume considerable area that is dedicated solely to caching and is therefore unavailable for extra computation or storage, increasing manufacturing costs. During operation, caches consume a large fraction of the total processor power, especially for low-power processors [26, 4]. Much of this power is used for tag checks and control rather than the actual instruction accesses [41]. Finally, caches are seldom used in real-time environments because they introduce unpredictable timing; unexpected cache misses can result in missed real-time deadlines [28].

As a result of these problems, many embedded processors and DSPs forgo hardware caches in favor of simpler, cheaper alternatives [11]. In fact, even some larger multi-core processors such as MIT's Raw [35] and IBM's Cell [16] use simple memories to reduce complexity and allow additional functional units to fit on a single chip. Some of these processors provide only a simple on-chip SRAM to hold all the program code and data (*e.g.,* the Texas Instruments TMS470, TMS320C28x and TMS320C000 family and the SPE of Cell). Others (such as the TI TMS320C24x and Analog Devices ADSP-21xx families) access external memory by default but provide a small on-chip *scratchpad* memory [3] that can be optionally accessed instead. Both of these are simple array memory structures that are directly addressed and explicitly managed by software. Compared to hardware caches of the same data capacity, they are about 20% smaller, have shorter access times and consume 20% to 50% less energy per access (assuming a 2-way set associative cache). These reductions are mainly due to their lack of tag storage and comparison structures. Access times are also easier to predict with SRAM and scratchpad memories since there is no possibility of a miss. 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 their code or data into manageable pieces and then manually copy the pieces into or out of the local memory

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS '06 October 21–25, 2006, San Jose, California, USA.
Copyright © 2006 ACM 1-59593-451-0/06/0010. . . $5.00
Reprinted from ASPLOS '06, Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems, October 21–25, 2006, San Jose, California, USA., pp. 293–302.

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.

This paper presents a complete software system that allows all or part of a simple memory, such as an SRAM or scratchpad, to be automatically managed as an instruction cache. This system frees the programmer from the burden of meticulously managing memory and provides performance similar to a hardware cache for processors where the extra area and design effort are prohibitive. In addition, there is the potential for lower power consumption and increased performance, relative to a hardware cache, due to customization of the software cache to each particular program's needs. The runtime portion of this system executes entirely on the core CPU (time-sliced with the user application), making it practical for use on a variety of existing architectures. Although the advantages and disadvantages of caches discussed previously apply equally to data and instruction caches, the two are accessed in very different manners and patterns. Most program code is easily analyzable and highly predictable while data accesses can be more complex and harder to statically analyze. While either type of cache can be implemented in software, they generally require very different analysis and optimizations. In this work, we have decided to focus on instruction caches exclusively. (See Section 5 for more on data caches.)

This work builds on earlier work in overlays and virtual memory by transferring the concepts to current architectures and incorporating modern code-caching optimizations. Using these techniques, we have effectively implemented a level-one instruction cache *entirely* in software. Of course, the primary challenge with implementing caches in software is achieving good performance. Operations that take a single cycle in a hardware cache (such as tag checks) can require many general-purpose instructions in software. To combat this, the number of expensive operations must be minimized. We find that with aggressive optimization, our software cache is able to achieve performance within 10% of a general-purpose hardware cache on many benchmarks.

This system is implemented on an actual processor and runs on real hardware. It is independent of the source programming language and has been used successfully with handwritten assembly code and C programs to date. We also provide full support for interrupts and allow arbitrary functions to be pinned down in the cache. The system is complete and robust enough that it is being used as a standard tool in several other independent research projects.

2. Design and Implementation

To explore the concept of software caching, we have implemented a working system on the Raw prototype microprocessor [35, 36]. Raw is one of a new generation of tiled processors and is composed of 16 processing cores (called *tiles*) arranged in a 2-D array and connected by communications networks. However, this work deals with only a single tile at a time. In isolation, each tile is similar to a typical embedded processor. A single tile contains a 32-bit, 8-stage RISC pipeline based on the MIPS instruction set with a 32 kB data cache and a 32 kB directly-addressed SRAM instruction memory (the *I-mem*). Instructions may only be fetched from the I-mem, not directly from external memory. External memory is accessed by sending messages over a special-purpose network to an external SDRAM controller.

In this paper, we adopt the usage of the terms "virtual address" and "physical address" from the early virtual memory literature. Addresses within the I-mem are referred to as *physical* addresses since they are the addresses understood directly by the 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 caching system.

2.1 Overview

The software instruction caching system consists of two components: a runtime library and a preprocessor program. The runtime library manages the cache during program execution. The preprocessor transforms the original program code (which assumes a 32-bit address space) into appropriate cache blocks and modifies control-flow instructions to use the runtime library.

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 unpredictable requests. The runtime library is responsible for servicing requests for instructions and managing the I-mem via normal processor instructions. 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 performs tasks such as rearranging code into convenient chunks and modifying code to make use of the runtime system.

After preprocessing, the modified object files are linked with the runtime library to create the final, cache-enabled binary. As a result, the I-cache system is actually integrated into each program. This means that different programs could potentially use completely different caching schemes. Although this can cause some difficulties when running in a multi-tasking environment, it is generally not a concern for embedded processors which typically run a single program at a time.

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 Code Modification

Since the I-caching system is integrated into each program, programs must be modified by a pre-processor in order to use it. 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, and during loading of the program. In our system, 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.

2.2.1 Cache Block Size

A *cache block* is defined as a chunk of program code that is retrieved, stored and evicted as a unit. It is similar to the concept of a cache line in a hardware cache. There are many possible ways to form blocks from the source program. Hardware caches simply chop up the program into fixed-size lines without regard for the program's structure or control-flow. The obvious choice for a software

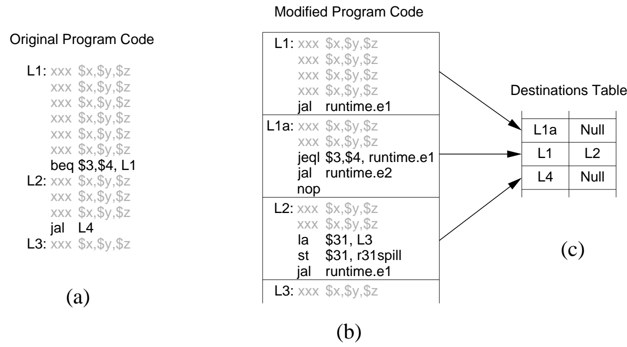

Figure 1. Example code modifications performed by the rewriter. The grayed-out instructions represent non-control-flow instructions. For illustrative purposes, a block size of 5 instructions is used here. Note that opcodes beginning with "b" use PC-relative addressing while opcodes beginning with "j" use absolute addressing.

instruction cache is to use the program's natural *basic blocks* as cache blocks. Because basic blocks execute sequentially until they reach their end, only the last instruction can cause control to leave the block. Also, if the cache loads one basic block at a time, only instructions that will actually be executed will be fetched. Each cache block in our system contains one basic block from the source program. However, to simplify bookkeeping each cache block has a fixed size. Therefore, basic blocks may need to be padded with `NOP` instructions or split into multiple blocks in order to fit the cache block size (see Fig. 1a,b). We have experimented with sizes of 8 and 16 instructions.

When using 8-word cache blocks, very large basic blocks need to be split many times, creating extra runtime overhead. To reduce this overhead, the rewriter may tag cache blocks with an *autoload* value that causes the runtime system to automatically load up to four consecutive cache blocks as a single unit. In effect, this allows the system to use cache blocks of 8, 16, 24 or 32 words.

2.2.2 Control-flow Instructions (CFIs)

The key to the operation of the software instruction cache is the modification of instructions that might cause control to pass out of the current cache block. These instructions are modified to jump to the runtime system instead of their original destinations (see Fig. 1b). The runtime system determines the originally intended destination, loads a new cache block into the cache if necessary, and transfers control to the appropriate block.

For maximum efficiency at run time, it may be necessary for the preprocessor to modify the addressing mode of CFIs in addition to their destinations. Because the runtime library remains resident in I-mem at a fixed location, any CFI that jumps to it should use an absolute addressing mode. This allows the CFI (and the block that contains it) to be placed anywhere within the I-mem and still jump to the same place. On the other hand, any CFI that jumps to a location within its own cache block should use a PC-relative addressing mode. Again, this will allow the cache block to execute correctly no matter where it is placed. By selecting the appropriate addressing mode during preprocessing, costly patching of destination addresses can be avoided at run time. The Raw instruction set contains both PC-relative and absolute conditional branches, allowing most instructions to be directly replaced (see the `beq` instruction in Fig. 1a and its replacement in Fig. 1b). If all the various combinations of CFI type and addressing mode are not available, they can be synthesized using sequences of other CFIs, albeit with a reduction in efficiency.

When CFIs are modified to point to the runtime system, their original destinations must be recorded. Therefore, the rewriter builds a table (the *destinations table*) that contains the original destinations for the CFIs that exit each cache block (see Fig. 1c). During run time, this table will be stored in DRAM along with the program code. The addresses stored in the table are virtual addresses, *i.e.,* they specify the location of the desired code in the external memory. The rewriter identifies all places where virtual addresses are used and converts them to physically-addressed calls to the runtime system. The runtime system can then use the data from the destinations table to fetch the appropriate code from DRAM.

Each cache block can have multiple exit addresses stored in the destinations table. To communicate which address should be fetched, the runtime system has multiple entry points. The rewriter modifies the different CFIs to jump to the appropriate entry point. However, some CFIs cannot have their addresses placed in the table because their destinations are not known until runtime. Indirect jumps get their destinations from a register and therefore require a special entry point in the runtime that expects the destination to be passed in a register. Additional entry points can be added for other special CFIs such as function calls or interrupt returns.

Our initial implementation had a special entry point for jump-and-link instructions that retrieved both the jump destination and the link address from the destinations table. However, we found that this made it difficult to apply certain optimizations to these jumps so a new approach was devised. The rewriter translates jump-and-link instructions into an explicit save of the link address, followed by a simple jump. Since the link address is virtual, it is independent of the physical address at which the block is loaded and can therefore be determined statically by the rewriter. This transformation eliminates a special case entry point from the runtime system and allows the jump to be optimized just as a regular jump would be. This technique can also be used for other compound instructions that cannot be implemented atomically under software I-caching.

2.3 Runtime System

2.3.1 Data Structures

Choosing appropriate bookkeeping data structures is crucial to the efficient operation of a software cache. Lookups must be very fast to ensure that cache hits have minimal overhead. Data structure size is just as important as speed because data structures should be stored in on-chip memory to ensure they can be accessed quickly. If stored in on-chip data memory, the bookkeeping structures will compete with the user program for space. Instead, we devote a portion of the I-mem to bookkeeping. Since the I-mem is already managed by the cache, this is a simple 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.

The primary function of the runtime system is to keep track of which blocks 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. Any chained (see Section 2.3.3) references to the block will still be able to use it but 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 of about 0.5 is used to reduce the number of conflicts.

In addition to the virtual-to-physical address mapping, the runtime system uses another table (the *block data table*) to store in-

formation about each block that is currently in the cache. Both this table and the hash table have a number of entries proportional to the total number of block storage slots within the I-mem. Because the cache block size is fixed, the number of storage slots is fixed and the size of all of the local data is static and independent of the size of the user program. This means that accessing and updating this data is fast and efficient and that the total application size is limited by the DRAM size, not the I-mem size.

2.3.2 Operation

When the runtime system receives control from a block, it finds the virtual address of the next block to be executed based on the runtime entry point to which the block jumped. Once the desired destination address has been found, a lookup is performed in the hash table and control is transferred to the corresponding physical address. If the destination address is not found in the table, requests for the block are sent to external memory. While the runtime system waits for the response, it selects a location for the new blocks and takes care of bookkeeping. When the response arrives the new code is copied into the appropriate place in the cache and its exit addresses (from the destinations table) are copied into the block data table for future use. 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 where any cache block may be placed in any slot within the cache. With a hardware cache, each cache line is typically limited to a small number of slots (between one and four) based on a hash of its address. If lines are pinned in a hardware cache, some of these slots become unavailable and the probability of thrashing increases significantly. In a fully-associative cache, pinning a block has a negligible effect on the placement of other blocks and thrashing is avoided. Furthermore, a two-way set associative cache (for example) permits only one line to be pinned in each set while a fully associative cache can pin as many blocks as will fit in the cache [18]. Since pinning code is crucial for meeting real-time deadlines, this is a valuable advantage for embedded systems.

If the cache is full, something must be evicted to make room. So far, we have implemented two different replacement policies: *FIFO* and *flush*. *FIFO* evicts the oldest block in the cache while *flush* 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 [19]; however, our system is not necessarily limited to these choices. (One benefit of implementing a cache in software is that it can be updated after the hardware is deployed, as additional improvements are made.)

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. FIFO has additional advantages over LRU and random when used with the chaining optimization described in the following section.

The flush policy is commonly employed by dynamic binary translators (see Section 5). It requires less bookkeeping than FIFO because blocks do not need to be individually evicted. The disadvantage of the flush policy is that a lot of very recently used code is evicted. However, as we will see later, the advantages of the flush policy are only truly realized when it is combined with chaining.

2.3.3 Chaining

With the system described above, every change in control-flow results in a call to the runtime system. Since even a hit in the runtime system takes about 40 cycles, the overhead is substantial and performance is poor. *Chaining* is an established technique [7, 39, 10, 2] that cuts out unnecessary jumps to the runtime system by modifying the code in the cache. When the runtime system loads a block into the cache, it goes back and changes the destination of the jump that requested the block so that it jumps directly to the new block. Now the next time that code is executed, it will skip the call to the runtime system and incur zero overhead (see Fig. 2). Chaining can be performed not only when a new block is loaded, but also when a block is requested that is already present in the cache. In fact, the runtime system can chain every time it is executed except when the original jump was indirect (*i.e.,* the target address was stored in a register) [7].

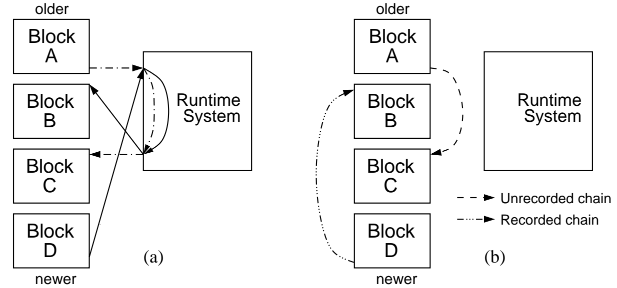

Figure 2. Example of jumps between blocks (a) before chaining and (b) after chaining. The chain from Block A to Block C does not need to be recorded when using FIFO replacement because A will be evicted before C.

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 jump that points to it must be changed back to a jump 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 jumps that have been modified to point to each block.

In general, there can be any number of chains to a particular block requiring a variable amount of storage for the unchaining information. However, to reduce the size and complexity of the data structures we statically allocate storage for only one chain per block. 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, as a side-effect of a FIFO replacement strategy, it is not necessary to keep track of chains that go from older blocks to newer blocks (see Fig. 2b). 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 that have been created are automatically thrown out. Thus, it is not necessary to keep track of or undo any chains. Now there 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. However, with FIFO it takes up to 24 additional cycles to determine whether a chain needs to be recorded and record it if it does.

Indirect Jump Optimization 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 [13] to accomplish this. The indirect jump is replaced with a sequence of instructions that compares the jump address to various individual addresses and then executes a normal jump if it finds a match. These normal jumps can then be chained as with any other jump. In our current implementation, this sequence is built up dynamically at runtime.

In essence, the address is being pre-screened to see if it matches a block that we have already seen. The drawback to this approach is that the pre-screening takes time and space. If an indirect jump goes to many different addresses, it can take longer to do all the individual comparisons than it would have to just call into the runtime. As a compromise, we adopt the heuristic that the sequence may grow only to fill any remaining space in the fixed-size cache block. However, if the rewriter sees that the empty space is below a threshold value, it will split the basic block and move the indirect jump to a new block. This ensures that a certain minimum number of individual comparisons can be done. When using 8-word cache blocks, moving the jump to its own block may still leave insufficient space. In this case, the rewriter can add additional empty cache blocks after the jump to meet the minimum. In our experience, guaranteeing space for at least 3 comparisons provides the best performance on most benchmarks.

Although this optimization of indirect jumps is possible with both the FIFO and flush policies, we have chosen to implement it only with the flush policy. This is 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 us to pursue more aggressive optimization techniques that might not be practical if they needed to be reversed.

2.4 Pinned code

A key feature for meeting real-time requirements in a cached environment is the ability to pin certain pieces of code in the cache. These pinned sections of code cannot be evicted and will therefore have consistent, predictable timing every time they are executed. Our system allows the programmer to specify functions that are to be pinned. These functions are stored in I-mem separately from the region used to store cache blocks. Therefore, the amount of space available for blocks is decreased. However, since the functions are permanently stored in I-mem, they require no additional runtime bookkeeping. Pinned functions are not modified by the rewriter so that they will not incur any I-caching overhead.

3. Evaluation

This section presents experimental results for a software-based I-cache system implemented on the Raw microprocessor. First, we describe our experimental methodology. Then, we present results on the performance of our software I-caching system. Next, we discuss energy consumption and give the results of our power estimation study. Finally, we briefly discuss the impact of using fixed-size cache blocks and present data on the amount of padding this requires.

3.1 Methodology

The system was evaluated using the Mediabench [23] benchmark suite. This set of nine benchmarks provides a sampling of communications and media applications that are important for the embedded domain. All data was collected using the Raw cycle-accurate simulator *BTL*. This simulator has been extensively validated against the actual microprocessor and models it precisely on a cycle-by-cycle basis [36]. Although our software instruction caching system works equally well on the simulator and the Raw chip, we use the simulator for this study because it provides a richer set of profiling tools to inspect the operation of our system. BTL also uses an idealized I/O model that provides very short, consistent times for I/O operations. This effectively removes any I/O effects from the results, thereby allowing us to focus on the user code.

The simulator also allows us to simulate alternative processor designs for comparison. In addition to simulating the actual Raw hardware, 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 two-way set associative cache with a 32 byte (8 word) line size and a 32 kB capacity. To create a fair comparison between the software system and the hardware cache, the size of the I-mem was increased by 25% for the software I-cache so that it would consume roughly the same die area as the hardware cache with its tag storage. 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 these benchmarks (except mesa) without requiring any form of caching.

To assess the energy consumption of our software I-cache, we use a version of our simulator which is adapted to work with Wattch [4]. 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 were adjusted to roughly approximate the power consumption of the actual microprocessor. CACTI [32] was used to generate the models for the hardware I-cache and SRAM memory as discussed below. In our models, the hardware I-cache accounts for about 25% of the total energy consumed.

3.2 Performance

Although attaining high performance was not the primary goal of this research, it is none-the-less an important factor. 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, particularly in the embedded domain. Performance is also related to energy consumption. The additional instructions that a software I-cache must execute to manage itself will require additional energy. Therefore, given a specific processor, reducing the number of instructions executed improves performance *and* decreases energy consumption.

To demonstrate the need for and effectiveness of the optimizations we have implemented, we present results from several different versions of the I-caching system. Fig. 3 shows the dramatic impact that even a moderate amount of chaining can have. The first column in each cluster represents the runtime of the benchmark using the general-purpose hardware cache model. The other results are normalized to this column for reference. The second column in each cluster is a baseline implementation that does not attempt to do any chaining and uses a 16-instruction cache block. Clearly, the performance of this version of the system is unacceptable since all of the benchmarks took between 3.5x and 8.5x longer than they would have with the hardware cache. The third column improves on the baseline system by chaining when a request hits in the cache. This version chains only normal branches and jumps, *i.e.,* it does not implement the optimizations for function calls and returns discussed earlier. We were initially concerned that the overhead of chaining and unchaining would be detrimental to performance and wanted to avoid creating chains that were unlikely to be used. The fact that a request has hit in the cache indicates that the requested block has been used at least once before and is therefore more likely to be requested again in the future. Although the improvements us-

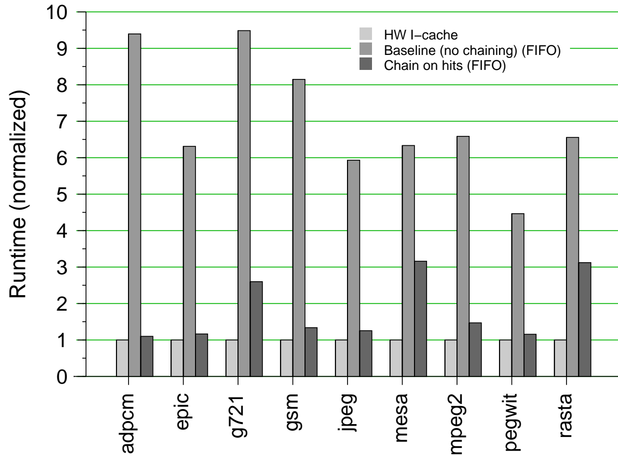

Figure 3. Performance impact of basic chaining. Run times are normalized to hardware I-cache performance.

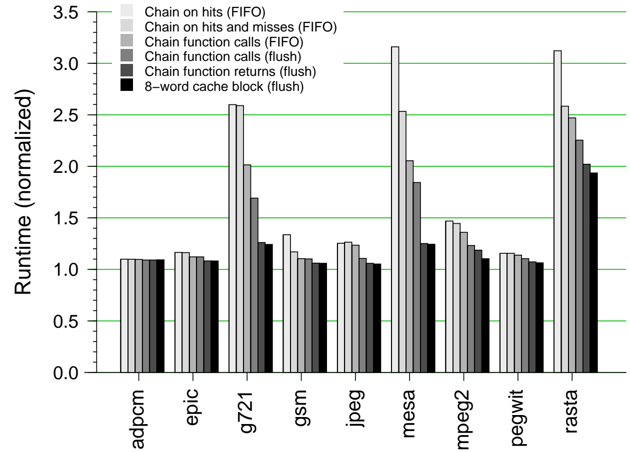

Figure 4. Performance improvement as optimizations are added. Run times are normalized to hardware I-cache performance.

	Hardware	Large I-mem	Software I-cache	
Benchmark	Cycles	Cycles	Cycles	Overhead
adpcm	11.19M	11.18M	12.21M	9.1%
epic	73.98M	73.93M	80.05M	8.2%
g721	367.1M	367.1M	456.1M	24.2%
gsm	91.75M	91.40M	97.19M	5.9%
jpeg	44.98M	44.79M	47.33M	5.2%
mesa	79.76M	—	99.23M	24.4%
mpeg2	1.646B	1.642B	1.816B	10.4%
pegwit	69.35M	69.24M	73.77M	6.4%
rasta	42.62M	40.78M	82.50M	93.6%

Table 1. Run time for Mediabench benchmarks in processor cycles. "Overhead" is the percentage of extra cycles relative to the hardware version.

ing this technique are impressive, three of the benchmarks still have very poor performance (150% or more slowdown) and the rest are only marginally acceptable (between 10% and 47% slowdown).

Fig. 4 shows the additional improvements obtained with additional optimizations. From these results it is clear that creating more chains is almost always a good idea, even if they may not be used. The first column in each cluster repeats the hit case chaining data from Fig. 3 for reference. The second column improves on the first by chaining on both hits and misses. In every case except jpeg, this produced better performance than chaining on hits alone, although for adpcm, epic, g721 and pegwit the effect was negligible. In jpeg, some of the chains created on misses filled the chain slot and prevented better chains from being created later. The third column shows the additional benefit realized when we changed the way we handle function calls (see Section 2.2.2) and were able to chain them as well. This change had a significant impact on every benchmark (except adpcm) but g721 and mesa showed exceptional improvement due to the enormous number of function calls they perform. This clearly demonstrates the importance of handling function calls efficiently.

Each of the versions discussed so far has used the FIFO replacement policy. As a result, there were some chains that could not be created due to the limited storage for unchaining information. The fourth column remedies this by switching to the flush replacement policy. Again, we see a substantial improvement in most benchmarks due to the additional chains that can be created and the reduced overhead for creating those chains. Jpeg and mpeg2 benefited greatly from the additional chaining while g721, mesa and rasta benefited most 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. The fifth column shows the results of optimizing function *returns* (*i.e.,* indirect jumps). The benchmarks that benefited from this the most were the same ones that benefited from chaining of function *calls*, namely g721 and mesa. However, additional profiling indicates that our pre-screening heuristic was only moderately effective on g721, pegwit and rasta, suggesting that additional improvements may be possible. Once again, the performance penalty of ignoring calls to (and the corresponding returns from) functions is clear.

Finally, the sixth column shows the performance when using 8-word (rather than 16-word) cache blocks. Initially, changing from 16-word to 8-word blocks hurt performance on all benchmarks. This was due to two factors. First, using smaller cache blocks required large basic blocks to be split more times. Each split introduces an extra jump instruction and causes separate calls to the runtime system. Second, changing to smaller blocks left less empty space to use for indirect jump optimization. No more than two pre-screening comparisons could be performed. To address these issues, this version also incorporates both the autoload feature and the extra blocks for indirect jump chaining described earlier. Because the smaller cache blocks waste less space (see Section 3.4), performance improved significantly on mpeg2 and rasta which have a large number of capacity misses. However, performance was essentially the same on g721, gsm, jpeg, mesa and pegwit and was actually slightly worse on adpcm and epic. For these applications, the relatively small gains in miss rate were offset by the increased overhead of the more complex runtime system.

Table 1 shows the data from the hardware I-cache, larger SRAM model and software system in tabular form. Because the software system can be changed for each program, the programmer is free to choose the version of the system that gives the best performance for a particular application. Therefore, the "software" column lists the value for the best variant on each benchmark. The large I-mem model provides a lower bound on the runtime since the program is stored entirely within the I-mem. The "Overhead" column is the percentage of extra cycles relative to the hardware cache model. In general, the hardware cache performed only slightly worse than the large I-mem. On five of the nine benchmarks (adpcm, epic, gsm, jpeg and pegwit), the software cache performed nearly as well as the hardware cache, incurring less than 10% overhead. G721 and mesa perform poorly due to their heavy use of function calls. Even with our optimizations, storing the link address and performing

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

indirect jump address comparisons take extra cycles. On these applications, more than half of the remaining overhead comes from these sources. Finally, rasta suffers from a very high miss rate because its working set is much larger than the available space. Under these circumstances, chaining is less effective because many blocks are evicted before their chains can be used.

3.3 Energy Consumption

Energy consumption is also an important consideration in the embedded domain. 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 our system, we compare it to a hardware instruction cache.

Previous studies [26, 4, 17, 40] have shown that instruction caches consume a substantial fraction of a modern processor's power. Data from actual processors [26, 4] as well as power estimation tools [17, 40] 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 [38, 32] 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 2) indicate that between 20% and 50% of the energy consumed by the caches would be eliminated when using equally sized SRAMs. Note that smaller caches have higher overhead because the tags are larger.

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.

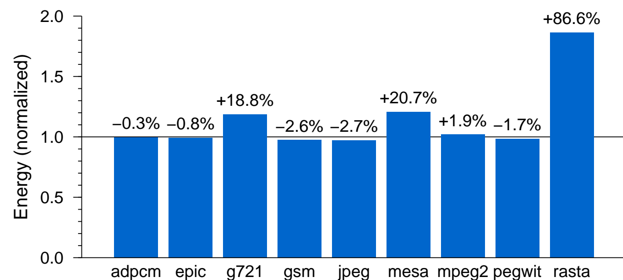

Figure 5. Total energy used to complete each benchmark with a software I-cache relative to a hardware I-cache.

To verify this 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 which is consistent with the values reported for actual processors in the literature [26, 4]. For the software cache case, the model for the cache is replaced with the SRAM model but everything else is left the same. For each benchmark, the version of the software system that gave the best performance was used. Fig. 5 shows the amount of energy used by the software I-cache normalized to the amount used by the hardware cache. The values on top of 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 four of the benchmarks, the software I-cache system actually consumes less energy than a hardware cache would have. However, it is also clear from the other benchmarks that high instruction overhead can dramatically increase the energy used.

3.4 Cache Block Padding

One factor that is contributing to the poor performance of rasta (and to a lesser extent mesa and mpeg2) is the use of a fixed cache block size. These benchmarks have larger code working sets resulting in higher miss rates. The problem is exacerbated by the padding that is inserted to make small basic blocks fill the fixed size cache blocks.

Previous studies have reported average basic block sizes of 6 to 7 instructions [39]. Because our system frequently adds one or two instructions to each block (to handle the fall-through case), we expected to see an average block size of 8 or 9. However, for 16-word cache blocks, we actually see an average of 5.75. This discrepancy may be due to splitting of blocks larger than 16 instructions that would have brought the average up. The net effect is that 60 to 65 percent of the instructions stored in the software cache are padding and only 35 to 40 percent of the cache is used for useful code. This is clearly a serious problem for larger benchmarks where cache capacity is a critical factor in performance.

Switching to 8-word cache blocks causes more splitting and reduces the average basic block size to 4.55 instructions. However, since the cache blocks are smaller, the amount of padding drops to about 40 percent and 60 percent of the instructions in the cache are useful. On the other hand, reducing the cache block size allows more blocks to fit in the cache and therefore requires more space for the block data and hash tables. This space must be taken from the space that was used to store cache blocks. This explains why we do not see a large improvement in performance, even though much less space is wasted on padding.

3.5 Discussion

Our results indicate that a software I-cache system can be a valuable tool for an embedded system. Performance on several of the benchmarks was comparable to both the hardware cache and larger I-mem. However, for many embedded processors, a hardware cache is not an option for the reasons we have mentioned before: additional die area, additional design cost, unpredictable timing. For these processors, we have created a tool that allows programmers to have the programming convenience and approximate performance of a hardware cache for non-critical portions of their code without giving up predictable timing on critical portions. We have also shown that this can be achieved while consuming less energy than a hardware cache would.

Because the system is implemented in software rather than hardware, it can be optimized for each individual application. For example, while the 8-word cache block version of the system produced the best performance on most of the benchmarks, the optimized 16-word version was better for adpcm and epic.

4. Future Work

There are still numerous opportunities to improve this work. Profiling results show that there are still a significant number of indirect jumps that fall through to the runtime system in some benchmarks. To address this, we plan to experiment with more sophisticated versions of the indirect jump chaining optimization. We may be able to optimize more of them by improving our heuristic or performing more analysis in the rewriter.

Based on our padding results, we can see that significant space is still being wasted in the cache. There are two approaches to reducing wasted space that we are considering. The first is to remove the requirement of fixed-size cache blocks and allow the variable sized blocks to be dense packed. Although this is feasible it would require extensive changes to the operation of the runtime system and rewriter. The second approach to reducing wasted space is packing multiple basic blocks into each cache block. Superblocks [20] (single entry, multiple exit non-looping regions of the control-flow graph) may be a good choice for this. Beyond that, we plan to experiment with including even larger portions of the control-flow graph in a single block, including loop structures. Not only does this have the potential to reduce wasted space, it could also eliminate calls to the runtime system. The effect would be the same as if all the jumps within the block were pre-chained. Of course, these two approaches could also be combined to yield even greater efficiency.

We would also like to look for additional ways to customize cache behavior to each individual program. One technique for doing this would be to incorporate profiling into the preprocessing phase. Because embedded applications typically have fairly regular behavior and predictable inputs, profiling can be an effective technique for identifying performance-critical regions of code. The regions 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.

In addition to working with conventional embedded architectures, we would like to explore the possibilities of making minor architectural changes to assist software-based caching. The challenge will be to devise efficient structures that maintain the reduced complexity and power consumption of a simple memory architecture while boosting performance or reducing the miss rate.

5. Related Work

This work has its roots in the early virtual memory work of the 1950's and 60's [9]. 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 swapped in from the secondary storage. This is analogous to the I-mem and external DRAM arrangement of our system. Before the development of hardware caches, the dominant strategies for implementing virtual instruction memory were overlays [30, 33] and segmentation [29]. Both of these systems work similarly to ours 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 our system, placing one or more entire procedures in each block. Segmentation systems 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. Our system 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 [14].

Earlier work in software caches has dealt primarily with hardware caches utilizing software miss handlers [6, 21, 18] or level two caches [24, 18]. Other techniques make automatic use of scratchpad memories [3, 11, 1] but they generally optimize only select portions of a program rather than providing a complete caching solution.

Systems like the VMP multiprocessor [6] and Jacob and Mudge's software managed address translation [21] have a hardware cache but employ a software cache-miss handler. VMP has hardware to check tags and handle cache hits but fires a special interrupt to invoke a software handler 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. In our all-software system, there is a single unified memory, allowing for a flexible partitioning of resources.

RAMPage [24] is an example of a system where some part of the memory hierarchy is under software control but not the level one 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 software managed. 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) [18] allows blocks to be loaded anywhere in the cache and uses a hash table to keep track of them as we do. However, it implements this hash table (and the lookups within it) in hardware and only invokes software for cache misses. Further, they do not attempt to run this software on the primary CPU but instead, use 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 conventional hardware.

With the growing prevalence of scratchpad memories [3] in embedded processors, several other techniques have been proposed to make automatic use of them. Most of these techniques focus on data [11] but several use scratchpad memory for code [1, 37, 34, 31]. 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. This work 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 static, off-line 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. This work handles all fetch decisions dynamically and can therefore adapt to unusual patterns or phased execution.

HotPages [27] is the data cache equivalent to this project. 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 2.3.3).

The class of applications called "dynamic binary translators" includes simulators (such as Mimic [25], Shade [7], Embra [39], DAISY [12] and DELI [10]), dynamic code generators (such as VCODE [15] and Java virtual machines) and run-time optimizers (such as Dynamo [2] and DynamoRIO [5, 22]). Simulators and dynamic code generators work by translating machine code (or virtual machine code) into the machine code of the host computer and running it. They typically do their translation at runtime so that they can avoid a separate translation pass, insert extra code for profiling or debugging [7, 39], or avoid translating code that is never run. Run-time optimizers attempt to modify a program while it's running to increase performance or security. All of these types of systems are frequently implemented using a *translation cache*. As code is translated, it is placed into the cache and run from there. The translation cache is, in essence, an instruction cache.

However, there are two major differences between these systems and a software-based 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 [8, 13]. Because of this, simulation and dynamic code generation 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 eviction policy and mechanisms play a much larger role in the software I-cache.

The second big difference is that an instruction cache only loads code into SRAM while a dynamic translator must also translate it. The extra overhead for translating a piece of code is substantial. DAISY spends an average of 4315 instructions doing translation of *each instruction* in the source program [12]. Because of this extra overhead, cache misses are proportionally 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.

That said, many of the core mechanisms in dynamic binary translators and software instruction caches are similar. The system we have developed could be used as a platform on which to build a simulator, virtual machine or dynamic optimizer.

6. Conclusions

In this paper we have demonstrated a software system that provides an automatically-managed level-one instruction cache for processors that lack special-purpose caching hardware. This system provides programming convenience, performance and energy consumption comparable to what would be expected with a hardware cache. It accomplishes this without giving up the ability to maintain predictable timing on critical portions of real-time programs.

Our results show that performance within 10% of a hardware cache is achievable on a variety of multimedia applications. The key to achieving this performance is the elimination of calls to the runtime portion of the system via chaining. In our experience, every possible opportunity to chain should be taken. The extra overhead of creating chains that may not be used is usually 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. We are optimistic that future work will narrow the gap between hardware and software caches even further and demonstrate conclusively that they can play a valuable role in embedded systems.

Acknowledgments

The first author would like to thank Volker Strumpen and Matthew Frank for their early advice and discussions and Paul Johnson for his assistance with the rewriter infrastructure.

This work was funded by DARPA, the National Science Foundation, and MIT's Project Oxygen.

References

[1] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. 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.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 1–12. ACM Press, 2000.

[3] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. 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, 2002.

[4] D. Brooks, V. Tiwari, and M. 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.

[5] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for Windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2000.

[6] D. R. Cheriton, G. A. Slavenburg, and P. D. Boyle. Software-controlled caches in the VMP multiprocessor. In *Proceedings of the 13th annual international symposium on Computer architecture*, pages 366–374. IEEE Computer Society Press, 1986.

[7] B. Cmelik and D. 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. ACM Press, 1994.

[8] R. F. Cmelik and D. 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.

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

[10] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. DELI: a new run-time control point. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 257–268, Nov 2002.

[11] A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing*, 1(4), 2005.

[12] K. Ebcioglu and E. 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, Jun 1997.

[13] K. Ebcioglu, E. R. Altman, M. Gschwind, and S. W. Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on Computers*, 50(6):529–548, 2001.

[14] A. E. Eichenberger, J. K. OBrien, K. M. OBrien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture. *IBM Systems Journal*, 45(1):59–84, January 2006.

[15] D. R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 160–170. ACM Press, 1996.

[16] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10–24, March-April 2006.

[17] S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, M. Kandemir, T. Li, and L. K. John. Using complete machine simulation for software power estimation: The SoftWatt approach. In *HPCA '02: Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, page 141, 2002.

[18] E. G. Hallnor and S. 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.

[19] K. Hazelwood and J. 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.

[20] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: an effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7(1-2):229–248, 1993.

[21] B. Jacob and T. Mudge. Software-managed address translation. In *HPCA '97: Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, pages 156–167, Feb 1997.

[22] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *USENIX Security Symposium*, San Francisco, August 2002.

[23] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, 1997.

[24] P. Machanick, P. Salverda, and L. Pompe. Hardware-software trade-offs in a direct Rambus implementation of the RAMpage memory hierarchy. *ACM SIGPLAN Notices*, 33(11):105–114, 1998.

[25] C. May. Mimic: A fast System/370 simulator. In *SIGPLAN '87: Papers of the Symposium on Interpreters and interpretive techniques*, pages 1–13, New York, NY, USA, 1987. ACM Press.

[26] J. Montanaro, R. T. Witek, K. Anne, A. J. Black, E. M. Cooper, D. W. Dobberpuhl, P. M. Donahue, J. Eno, G. W. Hoeppner, D. Kruckemyer, T. H. Lee, P. C. M. Lin, L. Madden, D. Murray, M. H. Pearce, S. Santhanam, K. J. Snyder, R. Stephany, and S. C. Thierauf. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *IEEE JSSC*, 31(11):1703–1714, November 1996.

[27] C. Moritz, M. Frank, W. Lee, and S. Amarasinghe. Hot pages: Software caching for Raw microprocessors. Technical Report LCS-TM-599, Massachusetts Institute of Technology Lab for Computer Science, 1999.

[28] H. Muller, D. May, J. Irwin, and D. Page. Novel caches for predictable computing. Technical Report CSTR-98-011, Department of Computer Science, University of Bristol, Oct 1998.

[29] P. 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.

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

[31] R. A. Ravindran, P. D. Nagarkar, G. S. Dasika, E. D. Marsman, R. M. Senger, S. A. Mahlke, and R. 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.

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

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

[34] S. Steinke, L. Wehmeyer, B. Lee, and P. 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.

[35] M. B. Taylor, J. Kim, J. E. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, Mar 2002.

[36] M. B. Taylor, W. Lee, J. E. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. 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, Jun 2004.

[37] M. Verma, L. Wehmeyer, and P. Marwedel. Dynamic overlay of scratchpad 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.

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

[39] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Measurement and Modeling of Computer Systems*, pages 68–79, 1996.

[40] S.-H. Yang, B. Falsafi, M. 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 Eighth International Symposium on High-Performance Computer Architecture*, pages 151–161, Feb 2002.

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