Tiny Instruction Caches For Low Power Embedded Systems

ANN GORDON-ROSS, SUSAN COTTERELL and FRANK VAHID*
University of California, Riverside

Instruction caches have traditionally been used to improve software performance. Recently, several tiny instruction cache designs, including filter caches and dynamic loop caches, have been proposed to instead reduce software power. We propose several new tiny instruction cache designs, including preloaded loop caches, and one-level and two-level hybrid dynamic/preloaded loop caches. We evaluate the existing and proposed designs on embedded system software benchmarks from both the Powerstone and MediaBench suites, on two different processor architectures, for a variety of different technologies. We show on average that filter caching achieves the best instruction fetch energy reductions of 60–80%, but at the cost of about 20% performance degradation, which could also affect overall energy savings. We show that dynamic loop caching gives good instruction fetch energy savings of about 30%, but that if a designer is able to profile a program, preloaded loop caching can more than double the savings. We describe automated methods for quickly determining the best loop cache configuration, methods useful in a core-based design flow.

Categories and Subject Descriptors: B.3.2 [**Hardware**]: Memory Structures—*design styles; cache memories.*

General Terms: Design.

Additional Key Words and Phrases: Loop cache, filter cache, instruction cache, architecture tuning, low power, low energy, fixed program, embedded systems.

1. INTRODUCTION

Reducing power of embedded processors is becoming increasingly important for mobile applications. Much of the dynamic power of a typical embedded processor is consumed by instruction fetching—for example, 30–50% in Lee et al. [1999a] and Segars [2001]—since instruction fetching happens on almost every cycle, involves switching of large numbers of high capacitance wires, and may involve access to a power hungry set-associative cache.

Thus, several approaches have been proposed in recent years to reduce instruction fetch power. Some have focused on encoding the address and data bus signals to reduce bus switching [Aghaghiri et al. 2001; Benini et al. 1998; Stan
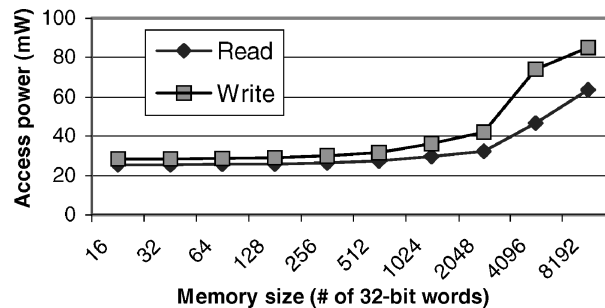
Fig. 1. Access power increase for different sized 32-bit wide memories in 0.18 micron CMOS technology.

and Burleson 1995]. Others have focused on compressing [Govindarajan et al. 2001; Ishihara and Yasuura 2000] or buffering instructions [Bajwa et al. 1997], also to reduce bus switching. Some work has looked at reducing the power of the cache itself by deactivating several ways of a set-associative cache when deactivation does not heavily impact performance [Albonesi 2000; Malik et al. 2000], or accessing items using multiple phases [Hasegawa et al. 1995], thus trading off performance for reduced power.

Another class of approaches adds an unusually small, perhaps 16 to 128 word (typically 64 to 512 byte) instruction cache, into the memory hierarchy. We will refer to such caches as *tiny* caches[1]. The extremely low power per access for a tiny cache is achieved in part due to very short wires inside the cache. For example, Figure 1 shows the increase in memory access power for 32-bit wide memories, as reported by the Artisan [n.d.] memory compiler for the UMC 0.18-micron technology. The x-axis shows the memory size, in terms of the number of 32-bit words. The y-axis shows the power per read or write access. Our own analysis using the CACTI model [Compaq n.d.] shows that for sizes under 128 words, the internal power is dominated by the sense amplifiers. At around 128 words is where the super-linearly increasing power due to the bitlines begins to dominate; hence the knee in the curve of Figure 1.

Another reason for very low power per access to a tiny cache comes from the fact that a tiny cache can be integrated very close to or even inside a microprocessor, resulting in shorter and hence lower power bus lines. For the M*CORE embedded processor architecture, an access to a tiny cache (of 32 bytes) consumed about 100 times less power than access to an 8 KB four-way set-associative L1 (level 1) cache [Lee et al. 1999a], using 0.25 micron CMOS technology at 1.7 V. Since many embedded applications spend much of their time in small loops that fit completely within a tiny cache, tiny caches can prove highly effective in reducing power.

One such tiny cache design was proposed by Kin et al. [1997] and was called a *filter cache*. A filter cache is a tiny direct-mapped cache introduced as the first level of memory (level 0) in the instruction memory hierarchy. A 256 byte filter

---

[1]Tiny caches were used to improve performance of direct mapped caches in Jouppi [1990].

cache was shown in Kin et al. [1997] to have a hit rate of between 60–85% on MediaBench benchmarks. Using a 32 KB direct-mapped cache for the L1 cache, the filter cache was shown to reduce instruction access power by over 50%, but at the expense of about 20% performance overhead. The energy*delay product related to memory accesses was reduced by about 50%. To reduce the performance overhead, Bellas et al. [1999] proposed using a profile-aware compiler to map frequent loops to a special address region recognized by the processor as loadable into the filter cache, resulting in less performance overhead along with improved energy savings.

Lee et al. [1999a, 1999b] and Moyer et al. [1999] proposed a different tiny cache, whose main distinguishing features from the filter cache are the absence of tags and misses. A filter cache, being a direct-mapped cache, stores part of the address of each instruction, and compares the tag with the desired address to determine a cache hit or miss. Upon a miss, a microprocessor stall occurs while the filter cache is filled from L1 cache. Tag accesses and comparisons require some energy overhead, while stalls due to misses cause performance and energy overhead. To eliminate the overheads, Lee proposed a tiny tagless instruction cache whose controller would transparently fill the tiny cache when a small loop was detected in the dynamic instruction flow. The controller detected the small loop by detecting a control of flow change caused by any branch instruction having a short negative offset, that is, a short backwards branch, or *sbb*, instruction. An sbb could be any existing branch instruction—an sbb was not a special or new instruction. A detected sbb would trigger the filling of the tiny cache, which Lee called a *loop cache*[2], during the next iteration of the small loop. The fill occurs nonintrusively as the processor continues to fetch and execute from regular instruction memory. Upon detecting the triggering sbb again, the loop cache controller would switch instruction fetching over to the loop cache. Fetches would continue from the loop cache until the triggering sbb was encountered but not taken, meaning the loop was being exited.

If the entire loop did not fit in the loop cache, then the cache would be filled completely with the first part of the loop, and fetching would switch back and forth between the loop cache and regular instruction memory.

Lee also explored a "warm-fill" version of the dynamic loop cache that continually filled the loop cache on every instruction fetch so that at any given time, the last N instructions were available in the cache, where N is the loop cache size. However, Lee et al. [1999a] showed that this design yielded little benefit—the power savings of being able to switch to loop cache fetching immediately after detecting an sbb did not outweigh the power overhead for keeping the loop cached filled.

Because filling of Lee's loop cache is nonintrusive (i.e., no microprocessor stall occurs), a control of flow change (cof)—that is, the case where the next instruction address is not the current address plus one—encountered within a loop would mean that the loop cache would not get filled with the entire loop. Thus,

---

[2]Bellas et al. [1999] also used the term "loop cache" to describe their profile-assisted filter cache approach.
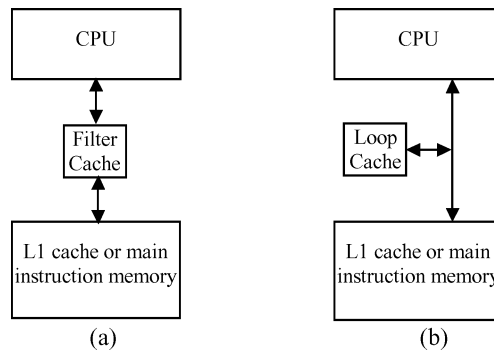
Fig. 2.   Memory hierarchy roles for (a) a filter cache (b) a loop cache.

in Lee's loop cache, a cof (other than the triggering sbb) encountered within a small loop would immediately terminate the loop cache filling or fetching. In other words, only loops with straight-line execution were supported.

Figure 2 illustrates the difference between a filter cache and a Lee-style loop cache with respect to their placement in the memory hierarchy. A filter cache is truly a first level of memory. In contrast, a loop cache is an alternative to the first level of memory, filling nonintrusively, and only accessed when a hit is guaranteed.

We refer to the Lee-style loop cache as a *dynamically loaded tagless loop cache*, or *dynamic loop cache* for short. The dynamic loop cache has two key advantages over the filter cache approach. First, no tag comparisons are necessary, thus reducing power per access. Second, performance is not degraded, since filling is done nonintrusively, and since there are no "misses," as the loop cache is only accessed when a hit is guaranteed. Some additional benefits include low power cache indexing since a simple wraparound counter can be used, and no need for the profile-aware compiler used in Bellas et al. [1999].

Lee reports instruction fetch reductions of 38% [Lee et al. 1999b] on Motorola's Powerstone benchmarks, using a loop cache of size 64 bytes (32 2-byte instructions). Larger loop caches yielded little further reduction, likely because programs typically do not have straight-line loops containing more than 32 instructions. The instruction fetch reduction translated to about a 15% reduction in total memory access power [Lee et al. 1999a] (considering both instruction and data accesses) for their M*CORE architecture.

In this paper, we introduce a *preloaded loop cache*, which is a variation of the dynamic loop cache. We also introduce a *one-level hybrid loop cache* and a *two-level hybrid loop cache*, each of which combines features of dynamic and preloaded loop caches. Furthermore, we highlight results of extensive experiments that compare all of the aforementioned tiny cache designs on a variety of benchmarks. We look at energy savings for all of the loop cache designs across a variety of different technologies. We also present a method to estimate the power savings of a loop cache configuration without the need to fully simulate a benchmark, producing comparable results in a greatly reduced amount of time.

2. CACHE EVAULATION ENVIRONMENT

We begin by describing the environment we used to evaluate the different tiny caches.

We considered, but rejected, two possible methods for evaluating a particular loop cache architecture. One method was through simulation of a system model using a hardware description language, followed by power analysis of the switching activity. However, we found that such simulation was too slow for our purposes. We sought to examine hundreds of different cache configurations for tens of benchmarks, meaning thousands of different instances. However, even the fastest hardware description language-based simulations and power analysis, utilizing behavioral level models, required nearly an hour per instance. The net result would have been several months of simulations, and no good foundation for fast future evaluations.

A second method we considered utilized a higher-level instruction-set simulation approach with a power-evaluation capability, such as Wattch [Brooks et al. 2000] (a power-extended version of SimpleScalar [Burger et al. 1996]). However, even such an approach required tens of minutes per instance, which would have required weeks of simulation. Even if this approach were fast enough, the approach would have two additional drawbacks. First, the approach would require us to rerun all examples if we wished to change the implementation technology. Second, the approach would limit us to evaluating results for just one particular processor.

We therefore developed a trace-based approach similar to that commonly used for traditional cache simulators like Dinero [Edler and Hill n.d.]. We ran each benchmark *once* on an instruction set simulator to generate an address trace for a particular microprocessor architecture. We developed a loop cache simulator, *lcsim*, that processes a trace file and counts the following loop cache related operations:

- *I-mem fetches*: the number of fetches the processor makes from regular instruction memory.
- *LC fills*: the number of writes to the loop cache.
- *LC fetches*: the number of reads from the loop cache.
- *LC detects*: the number of address comparisons made within the loop cache controller (necessary for loop cache extensions we will introduce).

We also modeled each loop cache controller in synthesizable VHDL and synthesized the controllers using the Synopsys Design Compiler [Synopsys n.d.] using the UMC 0.18 micron CMOS technology. However, rather than associating actual capacitance values with nets and letting power analysis tools output power per operation by computing switching times capacitance, we instead measured the average switching activity alone for each loop cache operation. Using this method, we could change the capacitance ratios to account for different technologies, without having to rerun *lcsim*. We determined the average capacitance of a net within the loop cache controller, and this capacitance served as a base value for which ratios were applied to determine the capacitance of internal buses with respect to the cache.

We used CACTI 3.0 [Compaq n.d.] to determine the energy per access for tiny caches of various sizes, utilizing a 0.18 micron technology. To determine the energy per access for a loop cache (which has no tag comparisons), we computed the energy for a direct mapped cache using CACTI, and subtracted the energy consumed by the tag comparisons. For the energy per access to main memory, rather than picking a particular memory and only providing results for that memory, we instead used different ratios so that different technologies (including future ones) could be explored.

Figure 2 shows the position of the loop cache and the filter cache in a memory hierarchy. We will refer to the first level of memory beyond the tiny cache, whether that first level memory is an L1 cache or just main instruction memory, simply as $L1$. For the loop cache, we are assuming that the access time to the loop cache and the access time to L1 is the same—1 clock cycle. The filter cache design assumes an access time of 1 clock cycle to the filter cache with a 4 cycle miss penalty (needed to fill the filter cache with a line), as assumed in Kin et al. [1997]. For our experiments, we assume no additional levels of cache.

For all cache designs, the cache size is reported as the number of instructions/entries that can fit in the cache. The size of each instruction is based on the instruction set architecture. For the MIPS processor, each instruction is 4 bytes, and for SimpleScalar, each instruction is 8 bytes.

3. DYNAMICALLY LOADED TAGLESS LOOP CACHING

3.1 Instruction Fetch Reduction Results

We first sought to examine the effectiveness of a dynamically loaded tagless loop cache for microprocessor architectures other than the M*CORE [Lee et al. 1999b], to see if similar results could be obtained. We examined the reduction in instruction memory fetches for a MIPS (32-bit) processor and for the SimpleScalar (64-bit, MIPS-based) processor [Burger et al. 1996]. We ran a subset of the Powerstone benchmarks [Malik et al. 2000] on the MIPS using the LCC compiler, and a subset of MediaBench benchmarks [Lee et al. 1997] on SimpleScalar using the gcc compiler, as shown in Table I. Results are shown in Figure 3.

The dynamic loop cache results in an average reduction in L1 accesses of about 30% for a loop cache with 32 instructions for the MIPS architecture, with almost no improvement for larger sizes. These results match the M*CORE results very closely. Our reduction of 30% is a bit less than the M*CORE reduction of 38%. The reason for the difference is that the M*CORE instruction set contains predicated instructions, which eliminate many branch statements in loops. Predicated instructions allow more loops to be supported by the dynamic loop cache, which requires straight-line execution of a loop body. Since the MIPS instruction set does not have predicated instructions, the loops compiled for the MIPS contained more internal cofs, and therefore less loops could be supported by the dynamic loop cache.

Figure 3 shows almost no loop cache savings for MediaBench on SimpleScalar. This lack of savings is due to the gcc compiler generating the loop exit condition check and branch at the beginning of most loops. The cof taken at the

Table I. Benchmark Descriptions

| Benchmark	Size of Assembly (Bytes)	Description
adpcm	7,648	Voice encoding
blit	4,180	Graphics application
compress	7,480	Data compression program
crc	4,248	Cyclic redundancy check
des	6,124	Data encryption standard
engine	4,440	Engine controller
epic*	154,016	Image compression
fir	4,232	FIR filtering
g3fax	4,384	Group three fax decode
g721*	95,024	Voice compression
jpeg	5,968	JPEG compression
jpeg decode*	355,072	JPEG compression
mpeg decode*	197,328	MPEG compression
rawcaudio*	199,920	Voice encoding
summin	4,144	Handwriting recognition
ucbqsort	4,848	U.C.B quick sort
v42	6,396	Modem encoding/decoding

*MediaBench.
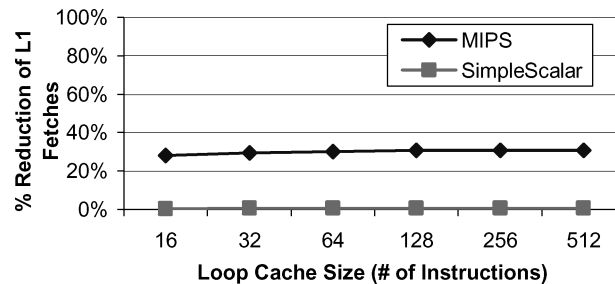
Fig. 3. Instruction memory access reductions for MIPS and SimpleScalar processors using a dynamically loaded tagless loop cache.

beginning of most loops terminates loop cache filling. Dynamic loop caching thus requires a compiler that generates straight-line loops. We also point out that we had to modify the default compiler flags that came with MediaBench, to turn off loop unrolling. Loop unrolling and loop caching are clearly not compatible.

3.2 Energy Savings Results

We computed energy savings of a dynamic loop cache, using the methods described in Section 2. Highlights of the data for each example are shown in Tables II and III. Table II first lists several statistics about the execution of each benchmark. *L1 fetches* is the number of reads from L1 cache, *LC fills* is the number of instructions written into the loop cache, *LC fetches* is the number of fetches from the loop cache, and *LC hit rate* is the percentage of instructions fetched from loop cache instead of L1. Note that adding the $L1$ and LC fetch numbers gives exactly the number of L1 fetches that would be required if there

Table II. Loop Cache Operation Statistics for Powerstone and MediaBench
Benchmarks Using a 128-Instruction Dynamic Loop Cache

Benchmark	L1 Fetches	LC Fills	LC Fetches	LC Hit Rate (%)
adpcm	42,013	6,573	21,878	34
blit	783	155	22,062	97
compress	124,622	23,779	13,951	10
crc	37,566	13,728	84	0
des	93,366	13,049	28,848	24
engine	324,435	37,478	86,172	21
fir	11,327	2,344	4,884	30
g3fax	447,317	24,843	680,706	60
jpeg	3,152,909	239,975	1,441,812	31
summin	849,295	422,094	1,060,492	56
ucbqsort	214,899	74,070	5,079	2
v42	2,298,426	474,389	144,125	6
epic*	95,299,112	18,585,747	39,748	0
g721*	558,814,268	66,478,034	5,733	0
jpegdecode*	7,972,925	243,607	175,868	2
mpegdecode*	419,972,712	36,785,194	8,259	0
rawcaudio*	16,372,340	295,832	4,702	0

*MediaBench.

Table III. Energy Savings for Powerstone and MediaBench Benchmarks Using a 128-Instruction
Dynamic Loop Cache

Benchmark	Energy (mJ)						Without LC (mJ)	Energy Savings (%)
	L1 Bus	L1 Access	LC Bus	LC Access	LC ctrl	Total		
adpcm	0.00	0.58	0.00	0.01	0.00	0.59	0.9	34
blit	0.00	0.01	0.00	0.00	0.00	0.02	0.3	95
compress	0.01	1.71	0.00	0.01	0.00	1.73	1.9	10
crc	0.00	0.52	0.00	0.00	0.00	0.52	0.5	0
des	0.00	1.28	0.00	0.01	0.00	1.30	1.7	23
engine	0.02	4.46	0.00	0.02	0.00	4.50	5.7	21
fir	0.00	0.16	0.00	0.00	0.00	0.16	0.2	30
g3fax	0.02	6.15	0.00	0.13	0.00	6.31	15.6	59
jpeg	0.16	43.38	0.01	0.31	0.00	43.86	63.4	31
summin	0.04	11.68	0.01	0.27	0.00	12.01	26.4	54
ucbqsort	0.01	2.96	0.00	0.01	0.00	2.98	3.0	2
v42	0.12	31.62	0.00	0.11	0.00	31.86	33.7	6
	Powerstone average							30
epic*	4.91	1,311.07	0.19	3.74	0.01	1,319.92	1,316.5	0
g721*	28.79	7,687.86	0.67	13.36	0.04	7,730.72	8,095.1	5
jpegdecode*	0.41	109.69	0.00	0.08	0.00	110.19	112.5	2
mpegdecode*	21.64	5,777.75	0.37	7.40	0.02	5,807.18	5,799.5	0
rawcaudio*	0.84	225.24	0.00	0.06	0.00	226.15	226.2	0
	Average							22

*MediaBench.

were no loop cache. Table III then lists energy consumed by various part of the
instruction memory hierarchy. *L1 bus* is the energy of the bus to L1 cache, *L1
access* is the energy of reading the L1 cache, *LC bus* is the energy of the bus
from the loop cache, *LC access* is the energy of writing and reading the loop

cache, *LC ctrl* is the energy of the loop cache controller, and *Total* is the sum of these energies, representing the total instruction fetch energy. Note that L1 accesses dominate the total energy. Finally, *without LC* is the instruction fetch energy with no loop cache, and *energy savings* is the percent reduction in energy obtained using the loop cache.

We see excellent savings for many of the Powerstone examples. Average overall energy savings for all benchmarks is 22%—lowered somewhat due to the low savings of MediaBench on SimpleScalar. Average savings of Powerstone alone was 30%.

3.3 Additional Experiments

We tried two dynamic loop cache variations, but they did not yield improvements over the basic loop cache. Each variation could be given a list of loop addresses that the loop cache controller would consider. In one variation, the controller blocked those loops from filling the loop cache. In the other variation, the controller would only fill the loop cache with those loops. The idea was to reduce unnecessary fills of the loop cache—that is, fills that were not followed by many fetches. However, neither variation showed significant improvement over the basic loop cache. The lesson learned was that minimizing the power within the loop cache itself is not a high priority, as that power is very small compared to L1 fetching. To really improve a loop cache, we need to increase the number of supported loops.

4. PRELOADED TAGLESS LOOP CACHING

4.1 Motivation

To improve upon dynamic loop caching, we examined the execution behavior of the Powerstone benchmarks, a high-level summary of which is in Table IV. We found that over 65% of the benchmarks' execution time was spent in loops of sizes less than 64 instructions. We also found that much of the remaining time was spent in small subroutines. Further details of our analysis can be found in [Villarreal et al.], as well as in [Villarreal et al. 2002].

Recall from Section 3 that a dynamically loaded tagless loop cache achieved at most a 30% L1 fetch reduction no matter how large the loop cache size. Through our analysis of the benchmarks, we found this reduction to be less than half of the potential reduction of 65%. The main reason for the difference is that a dynamic loop cache terminates loop cache filling or fetching when encountering a cof in a loop, but more than half of the loops contained such cofs. There are several reasons why a dynamic loop cache does not support cofs. First, filling the loop cache in the presence of branches is difficult, because we usually will not see all the loop instructions during an iteration of the loop, as the branches cause some loop code to be skipped. Recall that the controller is nonintrusive, and does not stall the processor to fill the loop cache, but rather just fills the cache from the dynamic instruction stream. Second, quickly detecting whether the cof exits the loop (i.e., jumps to an address outside the loop) is expensive in

Table IV. Loop and Subroutine Characteristics for Selected
Powerstone Benchmarks

Benchmark	Benchmark Characteristics
adpcm	Many small, frequent loops with no cofs. Very high frequency subroutines, some of which do not have frequent internal loops
compress	No high frequency loops with no cofs. High frequency subroutines and loops with cofs.
crc	No high frequency loops with no cofs. High frequency subroutines and loops with cofs.
des	Two large frequent loops—one with cofs and one without. High frequency subroutines where the majority of time is spent in internal loops.
engine	Five frequent loops—two with cofs, three without. Two very high frequency subroutines that contain all of these loops.
fir	Two frequent loops—one with cofs. A lot of frequent subroutines
g3fax	Two frequent loops with no cofs. Three frequent subroutines of which two contain frequent internal loops.
jpeg	Two very frequent subroutines—one with no internal loops, one with many internal loops. Many frequent loops both with and without cofs.
summin	Six frequent loops—three with cofs. Many subroutines, all with frequent internal loops.
ucbqsort	No loops without cofs. Four frequent subroutines—three with internal loops.
v42	Many frequent loops and subroutines. Only one loop with no cofs.

terms of power and hardware size. Third, supporting cofs within the loop cache makes using a simple wraparound counter for indexing into the loop cache much more difficult.

We also see from Table IV that the benchmarks utilize numerous subroutines—which are not supported by a dynamic loop cache. Sometimes those subroutines contain the most frequent loops, but sometimes those subroutines are actually called by such loops.

4.2 Solving the Fill Problem—Preloaded Loop Storage

We considered several possible solutions to the fill problem mentioned above. One possible solution was to stall the processor and fill the entire loop, but we sought to avoid stalls. A second solution was to allow loop cache misses, adding tags to the loop cache—this essentially reduces to a filter cache approach that is only filled and activated with small loops. Both these approaches involve some performance degradation—we leave them for future work.

We propose to solve the fill problem by storing the entire loop in the loop cache before the program begins executing, during the microprocessor boot sequence. Thus, once the program begins executing, the contents of the loop cache do not change—the loop cache is *preloaded*.

Preloading is possible in embedded systems, where the program executing on the processor is typically fixed, that is, the program does not change for the life of the system.

Obviously, this approach limits the number of loops that we can store in the loop cache. Thus, we only want to store the most frequently executed loops, meaning we must first profile the fixed program to detect those critical loops. We have found that many programs spend most of their time in two or three small loops [Villarreal et al. n.d.].

This approach has some parallels with the profile-guided compiler-based filter cache approach in [Bellas et al. 1999]. The key differences are that a preloaded loop cache is tagless and is missless.

4.3 Solving the Exit Problem—Exit Bits

A branch within a loop could jump to an address outside of the loop. Determining whether an instruction causes a loop exit the loop cache during runtime is rather costly. After an instruction is fetched, the microprocessor begins decoding and executing that instruction. During this time, we must begin comparing the target address of the branch instruction to the loop start and end addresses. However, we do not necessarily know where the target address lies within the instruction. There may be several forms of branch instructions, each with target addresses of different sizes and in different operand fields. As we do not wish to replicate the instruction decoder in the loop cache controller, we must instead wait for the microprocessor to provide us with the target address. However, if we begin the comparison at this point, the control lines may not be ready in time to meet the timing for the next instruction fetch, requiring the addition of extra cycles or the lengthening of the clock period—both highly undesirable. There are some workarounds, but these can be expensive in terms of size and power.

We propose a solution that involves pre-analysis of the loops before execution of the program—as we are preloading the loops, we may as well pre-analyze them too. We associate two extra bits with each word in the loop cache. Those bits will be used by the loop cache controller to quickly determine whether a cof should result in termination of loop cache operation:

- 00: means the instruction cannot possibly cause an exit from the loop. This encoding applies when the instruction is not a jump. The encoding also applies to an unconditional jump instruction whose target address is within the loop, and for a conditional jump instruction whose target is within the loop and whose next instruction is also within the loop.
- 01: means the instruction is a conditional jump that will exit the loop if the jump *is not taken*. Note that this is only possible for the last instruction in the loop.
- 10: means the instruction is a conditional jump that will exit the loop if the jump *is taken*.

We thus require that the microprocessor output signals to the loop cache controller indicating whenever a branch is executed or taken. The microprocessor

in the M*CORE dynamically loaded loop cache [Lee et al. 1999b] already implemented these signals.

Some branches have target addresses that we cannot determine statically, such as branches using indirect addressing or some form of register offset addressing. For these, we conservatively treat the target address as being outside the loop, giving such instructions an exit bit encoding of 10.

4.4 Indexing into the Loop Cache

A loop with straight-line code can be stepped through using a counter, as was done in the dynamic loop cache. Branches that jump to addresses within the loop pose a problem, though, as they would require the counter be loaded rather than incremented.

One simple solution to this problem is to eliminate the counter. Instead, given the loop's starting address M in memory and starting address C in the loop cache, we can derive the current loop cache address LPC from the processor's program counter (PC) as follows: $LPC = PC - M + C$. $PC - M$ represents the offset into the loop, which starts at C in the loop cache. We can precompute $-M + C$ to reduce the hardware needed during program execution. If we call that term the *offset*, then $LPC = PC + offset$. We will call this the PC offset approach. The drawback of this approach is increased power compared to a counter, due to the addition that must be performed.

We could also consider a counter approach to reduce power. We would initialize the counter to C. As long as instructions are not cofs, we would increment the counter after each loop cache fetch. However, when a cof occurs, we would update the counter with the target address T as follows: $counter = T - M + C$.

In the counter approach, we could conceptually disable the PC during loop cache operation. However, this requires additional modification of the microprocessor internals, as well as restoring the PC when leaving loop cache operation. We currently use the PC offset approach

4.5 Architecture

We now describe our basic preloaded loop cache, which incorporates all the features of preloaded loop storage, exit bits, and the PC offset indexing, in order to support a wider range of loops and subroutines.

The general structure of our basic preloaded loop cache is shown in Figure 4. The main components include *loop storage*, a *loop cache controller*, and *loop address registers* (LARs). The loop storage, like those in Bellas et al. [1999], Kin et al. [1997] and Lee et al. [1999b], is a very small basic memory, perhaps only 64 or 128 words. This small size enables tight integration with the microprocessor and very low power per access. Furthermore, the loop storage, like Lee et al. [1999b] and unlike Bellas et al. [1999] and Kin et al. [1997], contains no tag storage or tag comparison logic, to further reduce power. Instead, the loop cache controller is responsible for determining whether an instruction can be fetched from the loop storage. In contrast to a dynamically loaded loop cache, this storage holds multiple loops and subroutines simultaneously.
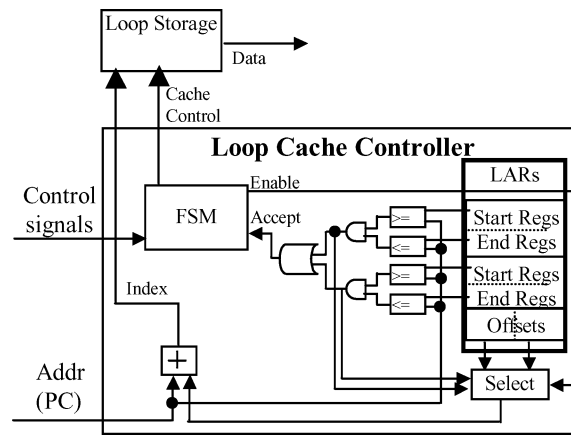
Fig. 4.   Architecture of a preloaded loop cache.

The loop address registers not only include the loop's start and end addresses in instruction memory, but also the loop's starting address in the loop storage—all of which are written into the registers during the program's boot sequence.

4.6 Operation

The preloaded loop cache controller is responsible for filling the loop cache, switching instruction fetches from program memory to the loop cache, and switching instruction fetches from the loop cache to program memory.

The state machine describing the controller is shown in Figure 5. The *Disabled* mode is the default state upon system reset. In the *Disabled* mode, the loop cache does not operate and does not store anything. All instruction fetches go to program memory. This state disables the comparators and loop address registers. If the loop storage, comparators and loop address registers can be put in a lower-power lossy state (to reduce static power consumption), then that occurs in this mode as well.

The controller transitions from *Disabled* to *Configure* when detecting a write to a loop address register. Each written loop address triggers a transition to the *ConfigFill* mode. In this mode, the loop cache is filled with instructions from the start to end address of the triggering loop address register. The controller includes a register to keep track of the next available address in the loop cache; this register is cleared in the *Disabled* mode, and then updated during the *ConfigFill* mode.

During this mode, the microprocessor must be suspended in order to enable the loop cache controller to generate instruction fetches. However, the microprocessor has not begun operation and thus this suspension does not affect runtime performance, but instead only increases the boot time (and even then, only slightly). We currently require that exit bits be written to a special register in the loop cache—they can be determined during pre-analysis and included in the boot routine. However, a microprocessor could be designed with a special loop cache fill mode that carries this out, by fetching and decoding and filling in exit bits, while executing the fetched instruction.
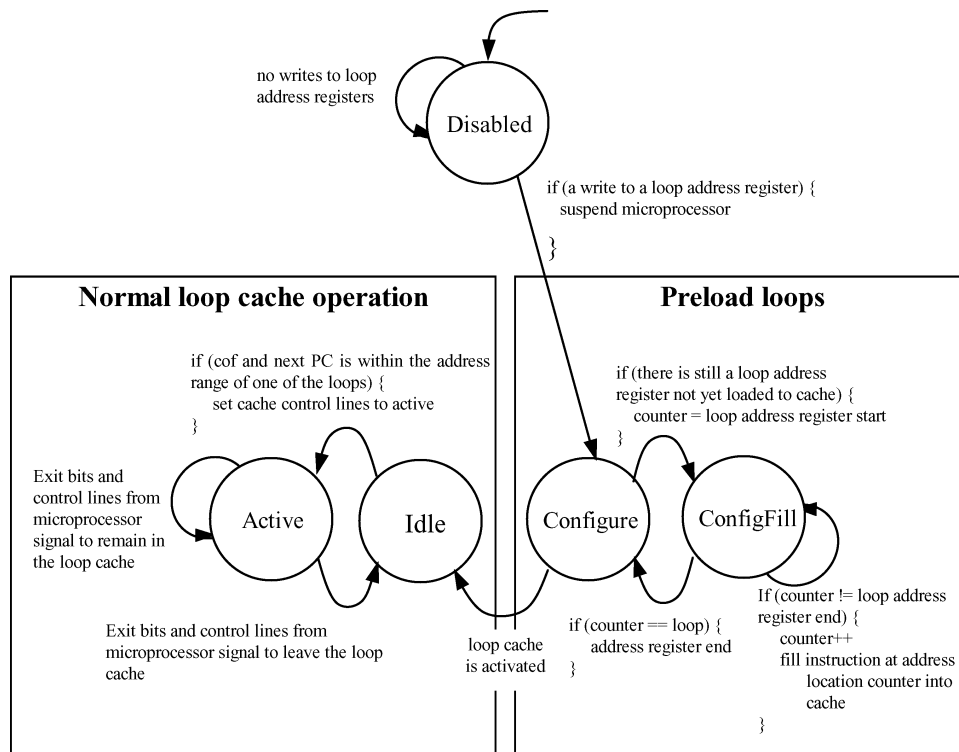
Fig. 5. Basic preloaded loop cache operation.

Note that we assume that all locations between a loop's starting address and ending address contain code, and in particular, not data. Most architectures do not mix code and data. For those that do, we require that at least the data do not appear inside of a loop selected for loop cache storage, since otherwise our exit bit determination could get confused by assuming that data are instructions. Also, note that since we use the microprocessor's fetch and decode logic, we can support variable length instructions.

The loop cache controller returns to the *Configure* mode after the *ConfigFill* mode finishes the filling the loop cache. The program's boot sequence may contain numerous iterations between *Configure* and *ConfigFill*, one per loop being stored in the loop cache.

The writing of the loop address registers and the filling of the loop storage take place during the microprocessor boot sequence. We assume part of the boot sequence includes setting values for all of the configurable features of the microprocessor architecture, including values related to voltage level, cache way enable/disable, and loop addresses and exit bits for a preloaded loop cache. These values can be stored in a configuration routine called as part of the boot sequence. Thus, the configuration methodology we propose does require recompilation of the program after profiling determines the best configuration values, which we insert into the configuration routine; however, note that this

methodology still uses a standard compiler and in particular does not require a special compiler.

From the *Configure* mode, the controller transitions to the *Idle* mode when detecting the setting of the activate flag in a loop cache configuration register. The active flag simply enables the loop cache controller. In the *Idle* mode, fetches continue to occur from program memory. However, the controller monitors the address bus and on every cof, compares the next address with the loop address registers checking for a transition to the *Active* state. Because a preloaded loop cache supports subroutines, the controller cannot simply look for a start address or end address—due to some code jumping directly to a statement within a subroutine. Instead, the controller must check if the target of a cof is between the start and end addresses of all the preloaded loops.

If a match is found, the controller stores the loop's starting memory address and starting loop storage address, and transitions to *Active* mode. In *Active* mode, the controller disables the bus between the microprocessor and instruction memory, and instead fetches instructions from the loop cache. The address of the instruction in the loop cache is obtained using the equation described in Section 4.4.

4.7 Choosing the Loops to Preload

To select regions to put into the preloaded loop cache, we developed an approach for detecting the best loops/subroutines to include. We developed the loop analysis tool LOOAN [Villarreal et al. n.d.] for the MIPS and SimpleScalar architectures. The tool parses an assembly file and detects loops. A subroutine map file is also read, which is output by many compilers. Knowing the locations of all loops and subroutines, the tool parses a trace file and outputs statistics on the percentage of overall execution time spent in each loop/subroutine, the number of visits to each loop/subroutine, the iterations per visit, and so on. We could sort the regions by percentage of overall execution, but that does not take into consideration the size of the region. Some regions are so large that even with a few iterations, the overall execution time is very large. Since storage in the preloaded loop cache is limited, we seek to preload small regions of code that have a large percentage of execution time. To achieve this, we sort the loops in decreasing order by a metric that we refer to as the *execution density* of a region. The execution density is the ratio of execution time over size.

Determining which loops to preload is equivalent to the fractional knapsack problem. We take the region with the highest execution density and place as much of that loop as possible into the preloaded loop cache—recall from Section 1 that if a loop does not fit completely into a loop cache, we can just load the first part of the loop. A region is ignored if the region includes a subregion that has already been captured, or if the region is a subregion of a previously captured region of code. This packing of loops into the loop cache continues until the loop cache is filled or the maximum number of regions that can be stored is reached.
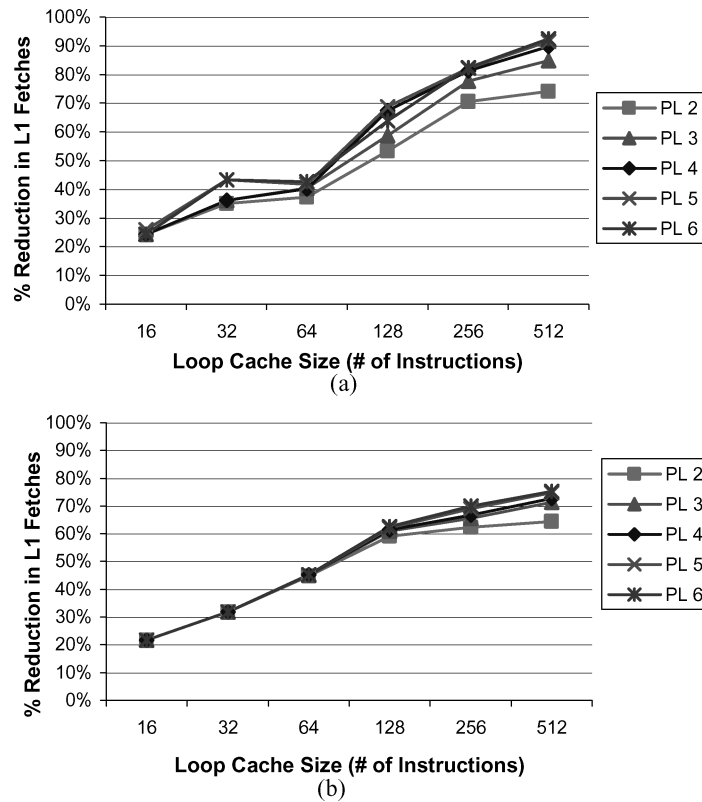
Fig. 6. Average instruction memory fetch reductions for the preloaded loop cache with two to six preloaded loops (PL two to PL six) running: (a) Powerstone on MIPS and (b) MediaBench on SimpleScalar.

4.8 Instruction Fetch Reduction Results

The L1 fetch reductions due to the preloaded loop cache for Powerstone on a MIPS can be seen in Figure 6(a), and for MediaBench on SimpleScalar in Figure 6(b). We tested preloaded loop caches of sizes ranging from 16 to 512 instructions with two to six preloaded loops. We see steep improvements as the loop cache size is increased up to about 256 instructions, where the improvement begins to level off. We also note that most of the reductions come from the first 2–3 loops.

For Powerstone on a MIPS, the L1 fetch reduction is as large as 92% for a loop cache of size 512 instructions with six preloaded regions of code. This makes sense since most programs spent most of their time in a small amount of code. Recall that the dynamic loop cache did not achieve more than 30% L1 fetch reduction regardless of the loop cache size, due to lack of support for cofs.

A 512-instruction cache is a 2 KB cache, beginning to approach the size of a typical L1 cache, where 8 KB to 32 KB sizes are common. Recall from

Table V. Loop Cache Operation Statistics Powerstone and MediaBench Benchmarks Using a 128-Instruction Preloaded Loop Cache Supporting up to six loops

Benchmarks	L1 Fetches	LC Detects	LC Fetches	LC Hit Rate (%)
adpcm	26,341	15,984	37,550	59
blit	229	510	22,616	99
compress	84,332	45,384	54,241	39
crc	306	3,678	37,344	99
des	54,577	6,150	67,637	55
engine	120,955	151,902	289,652	71
fir	3,960	6,234	12,251	76
g3fax	224,291	262,506	903,732	80
jpeg	3,601,272	1,156,248	993,449	22
summin	953	121,920	1,908,834	100
ucbqsort	20,952	183,174	199,026	90
v42	1,541,359	1,631,382	901,192	37
epic*	42,677,355	51,385,950	52,661,505	55
g721*	217,948,967	78,319,584	340,871,034	61
jpegdecode*	5,061,161	529,137	3,087,632	38
mpegdecode*	93,339,867	69,964,435	326,641,104	78
rawcaudio*	2,678,225	446,136	13,698,817	84

*MediaBench.

Figure 1 that power per access begins to increase steeply at a size around 128 or 256 instructions. For smaller loop cache sizes of 128 and 256 instructions, L1 fetch reductions for Powerstone were around 60% and 75%, respectively—still significantly better than a dynamic loop cache.

For MediaBench programs, whose loops tend to be somewhat larger, L1 fetch reductions for loop caches sizes of 128 and 256 instructions were around 45% and 75%, respectively. Recall that a dynamic loop cache did not reduce L1 fetches for MediaBench on SimpleScalar, since the loops all began with cofs.

4.9 Energy Savings Results

The loop cache operation statistics are shown in Table V, and energy results are shown in Table VI. The column headings are the same as Table II, except that the statistic *LC fills* is not shown since a preloaded loop cache does not get filled during program execution, and that *LC detects* has been added, indicating the number of address comparison operations performed by the loop cache controller.

We see much improved loop cache hit rates, especially for MediaBench on SimpleScalar, but also for Powerstone on MIPS, resulting in an average instruction fetch energy savings of 66%. This savings is triple that of the dynamic loop cache on all the benchmarks, and more than double that of the dynamic loop cache on the Powerstone benchmarks alone.

Figure 7 compares the energy savings of the dynamic and preloaded loop caches on each example for a 128-instruction loop cache. We see that the preloaded loop cache is superior to every benchmark.

Table VI. Energy Savings for Powerstone and MediaBench Benchmarks Using a 128-Instruction
Preloaded Loop Cache Supporting up to six loops

| Benchmark	Energy (mJ)						Energy without LC (mJ)	Energy Savings (%)
	L1 Bus	L1 Access	LC Bus	LC Access	LC ctrl	Total		
adpcm	0.00	0.36	0.00	0.01	0.00	0.37	0.9	58
blit	0.00	0.00	0.00	0.00	0.00	0.01	0.3	98
compress	0.00	1.16	0.00	0.01	0.00	1.17	1.9	39
crc	0.00	0.00	0.00	0.01	0.00	0.01	0.5	98
des	0.00	0.75	0.00	0.01	0.00	0.77	1.7	55
engine	0.01	1.66	0.00	0.05	0.00	1.73	5.7	70
fir	0.00	0.05	0.00	0.00	0.00	0.06	0.2	75
g3fax	0.01	3.09	0.00	0.17	0.00	3.27	15.6	79
jpeg	0.19	49.54	0.00	0.18	0.00	49.92	63.4	21
summin	0.00	0.01	0.01	0.35	0.00	0.38	26.4	99
ucbqsort	0.00	0.29	0.00	0.04	0.00	0.33	3.0	89
v42	0.08	21.21	0.00	0.17	0.00	21.46	33.7	36
epic*	2.20	587.13	0.53	10.59	0.08	600.53	1,316.5	54
g721*	11.23	2,998.42	3.42	68.52	0.33	3,081.92	8,095.1	62
jpegdecode*	0.26	69.63	0.03	0.62	0.00	70.54	112.5	37
mpegdecode*	4.81	1,284.12	3.28	65.66	0.31	1,358.18	5,799.5	77
rawcaudio*	0.14	36.85	0.14	2.75	0.01	39.89	226.2	82
							Average	66

*MediaBench.
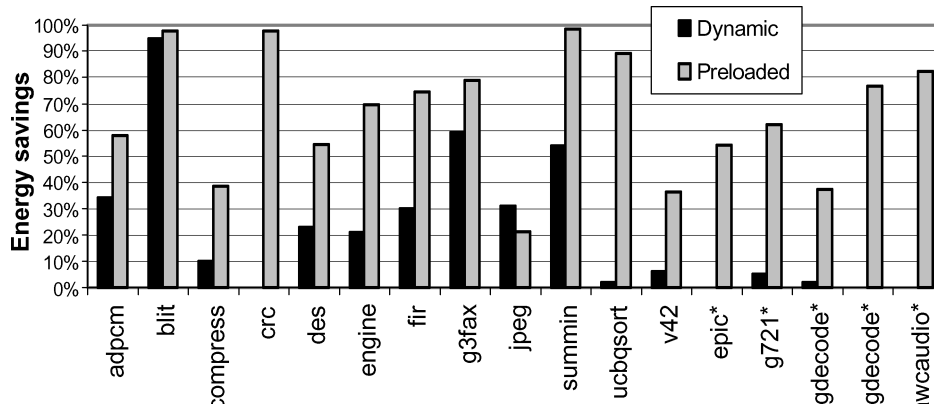
Fig. 7.   Percentage of instruction fetch energy saved by dynamic and preloaded caches on Power-
stone and MediaBench (*) programs running on a MIPS or SimpleScalar architecture, respectively,
using a 128-instruction loop cache.

5. COMBINING THE DYNAMIC AND PRELOADED TAGLESS LOOP CACHES—HYBRID LOOP CACHING

5.1 Motivation

Technology constraints could in some cases limit us to a loop cache smaller
than 128 instructions. For example, Lee found that for a particular 0.25
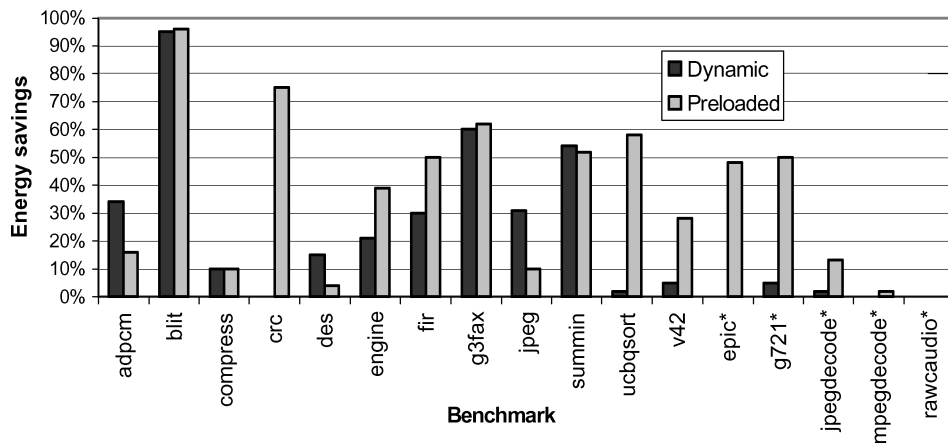micron, 1.75 V technology, a 16-instruction loop cache was energy optimal.

Fig. 8. Percentage of instruction fetch energy saved by dynamic and preloaded caches on Power-stone and MediaBench (*) programs running on a MIPS or SimpleScalar architecture, respectively, using a 32-instruction loop cache.

A 32-instruction loop cache had slightly worse energy, while a 64-entry loop cache had significantly worse energy.

Figure 8 compares the dynamic and preloaded loop caches on our benchmarks (for the same technology as reported in the previous section), this time utilizing a 32-instruction loop cache. We see that sometimes a dynamic loop cache is better, while sometimes a preloaded loop cache is better. This data motivated us to design a loop cache that could behave as either a dynamic loop cache, preloaded loop cache, or both—a *hybrid* loop cache. We considered two possible hybrid loop caching—a one-level scheme and a two-level scheme.

5.2 One-Level Hybrid Loop Cache

In the one-level hybrid loop cache, part of the loop storage is reserved for dynamic loop caching, while the rest of the loop storage holds preloaded loops. The preloaded part thus need not store loops that are supported by dynamic loop caching.

5.3 Two-Level Hybrid Loop Cache

In the two-level hybrid loop cache, a second level of loop storage is added, which stores preloaded loops. The first level loop storage by default acts like a dynamic loop cache. However, if loops are preloaded into the second level (along with start/end addresses in the loop address registers and the exit bits), then if a cof occurs, the target address is checked against the loop address registers. If a match occurs, then the first level loop storage is filled with the appropriate loop from the second level. Thus, the first level loop storage could be filled from L1 when a short backwards branch is detected, or from the second level loop storage when an address in a preloaded range is detected. In either case, once filled, fetching switches over to the first level loop storage. Fetching terminates
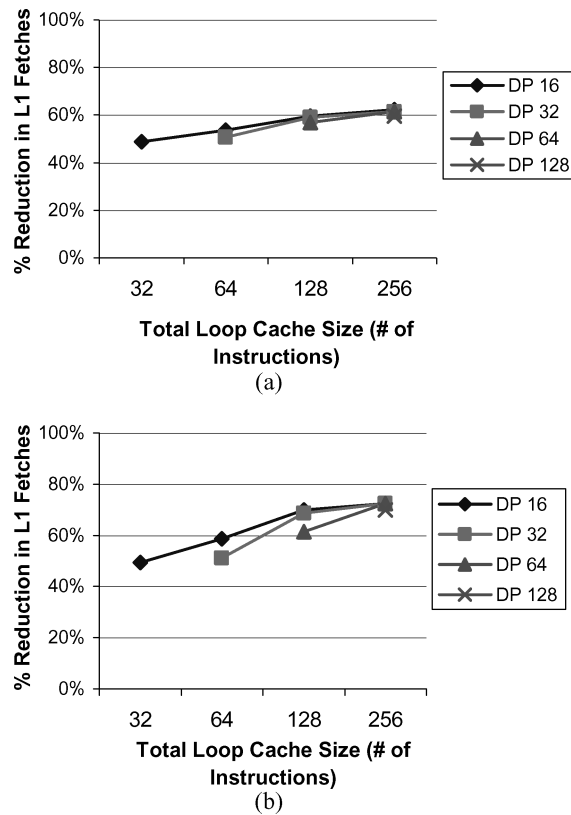
Fig. 9.   Instruction memory fetch reductions for the one-level hybrid loop cache supporting up to six preloaded loops, with increasing dynamic partition sizes (DP) running: (a) Powerstone on MIPS and (b) MediaBench on SimpleScalar.

if a cof is encountered while in dynamic loop cache mode, or based on a cof and exit bits while in preloaded loop cache mode.

5.4 Results

Figure 9 shows results for both the Powerstone and the MediaBench benchmarks for the one-level hybrid loop cache. We examined total loop cache sizes ranging from 32 to 128 instructions, with dynamic portions starting at a size of 16 entries and not exceeding one half of the total loop cache size. Our results show that even with a very small total cache size of 32 instructions with a dynamic portion of 16 instructions, the one-level hybrid loop cache reduces L1 fetching by 50% for both benchmark suites (the dynamic or preloaded caches each reduced L1 fetches by 30% to 40% for a size of 32).

We also see that for each total cache size, the largest reduction in L1 fetching occurs with a dynamic portion of 16 instructions, following closely the results of a purely dynamic loop cache. Also, the L1 fetch reduction tops off at approximately 60% and 70% for Powerstone and MediaBench, respectively, with a total cache size of 128 instructions and a dynamic portion of size 16.
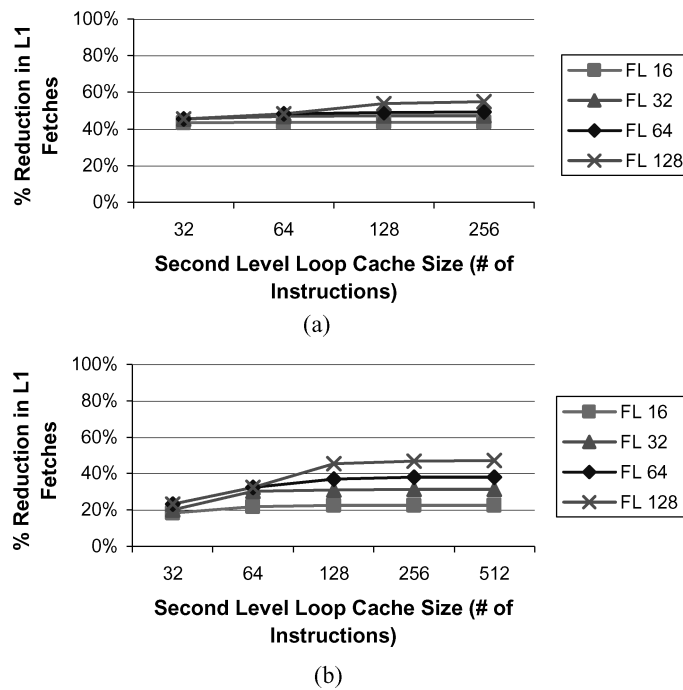
(a)

(b)

Fig. 10.   Instruction memory fetch reductions for a two-level hybrid loop cache with first level (FL) sizes from 16 to 128 instructions, running: (a) Powerstone on the MIPS and (b) MediaBench on SimpleScalar.

Table VII. Average instruction fetch energy savings for Powerstone running on MIPS for a dynamically loaded loop cache, and one-level and two-level hybrid loop caches with no preloaded loops

	Main Loop Cache Size (%)			
	16	32	64	128
Dynamic	30	30	30	29
One-level hybrid	30	30	30	29
Two-level hybrid	29	29	29	28

Results for the two-level hybrid loop cache design can be seen in Figure 10 for both the Powerstone and MediaBench benchmark suites, with first level (FL) sizes ranging from 16 to 128 instructions and second level sizes ranging from 32 to 256 instructions. We see that, for the benchmarks studies, the two level scheme does not seem to perform as well as the other loop caching methods.

A nice feature of both hybrid loop caching schemes is that they can be used as a dynamic loop cache only, if desired. This feature is useful in case the designer does not wish to or is not able to preload loops. Table VII shows that the instruction fetch reductions for a hybrid cache operating as a dynamic loop cache only are essentially the same as the reductions for a pure dynamic loop cache.
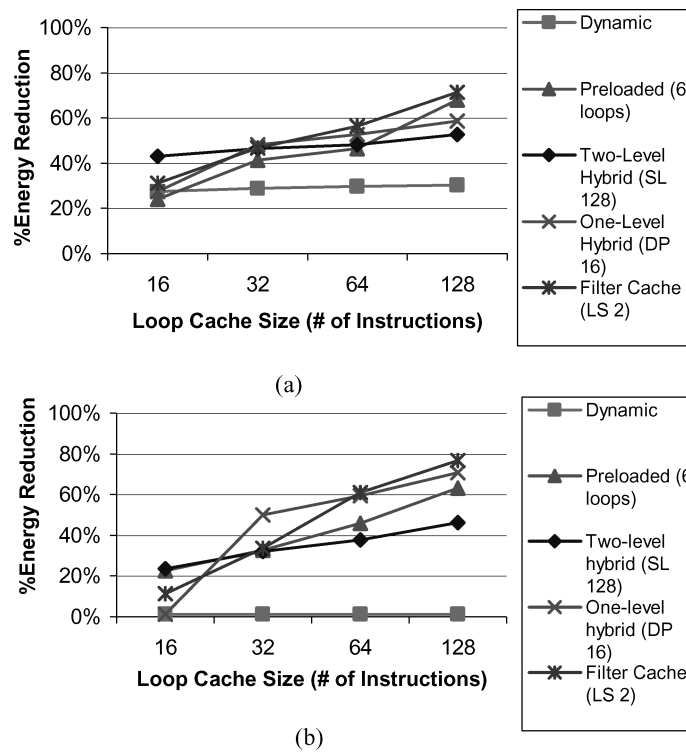
(a)

(b)

Fig. 11. Percentage of instruction fetch energy saved by a dynamically loaded loop cache, a preloaded loop cache with six loops, a two-level hybrid loop cache with a second level of 128 entries, a one-level hybrid cache with a dynamic portion of 16 entries, and a filter cache with a line size of two instructions for (a) Powerstone on MIPS and (b) MediaBench on SimpleScalar, for different (first-level) loop cache sizes.

6. COMPARING TINY CACHE ENERGY SAVINGS

We now compare the above-mentioned loop caches to a more traditional tiny cache (i.e., a filter cache). Figure 11 shows the average instruction fetch energy reductions for the Powerstone and the MediaBench benchmarks. We see that for a small loop cache of size 32 instructions, a one-level hybrid loop cache works best. However, for larger loop caches, a filter cache seems to be the best option. Not only does the filter cache save the most energy, but the filter cache also requires no preloading. Compared to the other tiny cache requiring no preloading, namely, a dynamic loop cache, the filter cache achieves more than double the energy savings (for Powerstone).

However, recall that one of the advantages of loop caching over filter caching was the absence of misses, meaning no performance overhead. Figure 12 provides the average performance penalty incurred for different sizes of filter caches. Because a filter cache's line size can have a significant impact on hit rate, we provide data for different line sizes too. We see very significant performance penalties incurred by the filter cache, with the lowest penalty still being 20%.
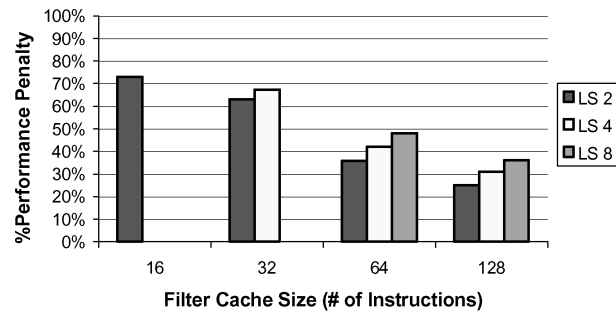
(a)
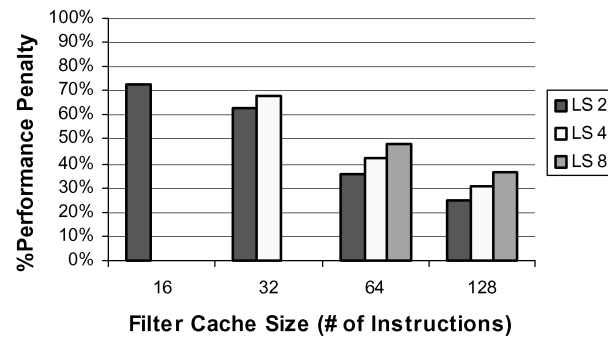
(b)

Fig. 12.   Average performance penalties for the filter cache with varying line sizes (LS) assuming a four cycle miss penalty for (a) Powerstone on MIPS and (b) MediaBench on SimpleScalar.

This performance penalty impacts energy. The data in Figure 11 *only consider instruction fetch energy*. That data do *not* consider the energy of other components of a system—such as the energy of the microprocessor itself, energy of the clock distribution network, or the static energy consumption of the L1 cache. During filter cache stalls, those other components will be consuming energy. The energy consumed by those other components varies greatly among different systems, so we do not present that particular data here—but we will be reporting the overall impact of various L1 and tiny caching methods across a range of systems in future work. The extra energy of those other components due to misses likely outweighs the instruction fetch energy savings of a filter cache.

Among the loop caching methods, we conclude that the one-level hybrid loop cache is likely the best choice. The one-level hybrid achieves far greater savings than a pure dynamic loop cache. The one-level hybrid also performs essentially the same (slightly better) than a preloaded loop cache, but has the advantage of being able to operate in a dynamic-only mode—in case the designer does not preload loops.

7. CONSIDERING DIFFERENT TECHNOLOGIES

We sought to determine the dependency of energy savings on the particular implementation technology as well as layout styles. A key difference among

16 Instruction Cache

32 Instruction Cache

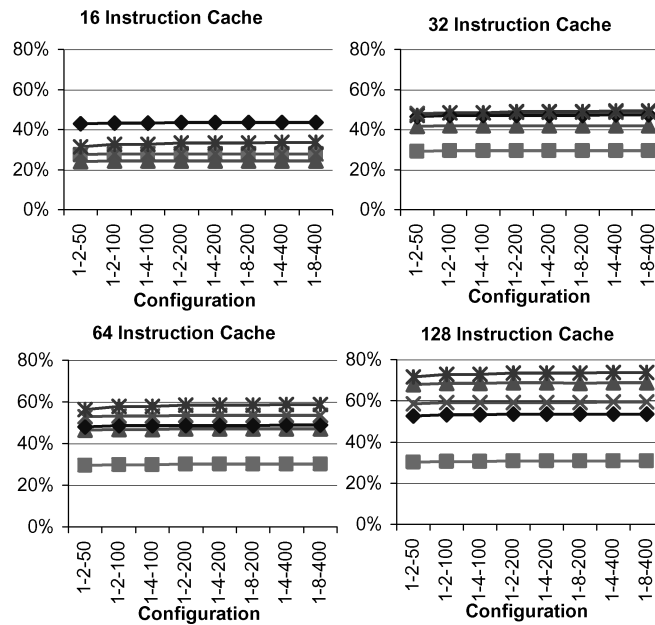**64 Instruction Cache**

128 Instruction Cache

Fig. 13. Instruction access energy savings for various power ratios for Powerstone running on MIPS (see Figure 11 for legend).

technologies and styles is the power ratio of L1 accesses to internal net switching in the controller, and of loop cache access (fill or fetch) to that of internal switching. In general, deep submicron technologies tend to increase this ratio [ITRS 1999]. We thus considered a number of ratios of increasing magnitude, ranging from 50:1 [Sias et al. 2001] to 400:1 for L1 access to internal net ratio, and from 2:1 to 8:1 for loop cache access to internal net ratio.

Results are shown in Figure 13 for Powerstone running on the MIPS and in Figure 14 for MediaBench running on SimpleScalar. The y-axis shows the percent instruction fetch energy savings compared to no loop cache. The x-axis lists the ratios for internal net power to loop cache access and to L1 access—so 1:2:50 means a loop cache access net is twice as power costly as an internal net, and an L1 net is 50 times as power costly as an internal net. 1:2:50 was the ratio we used in all the earlier data. While we did observe some fluctuation in total instruction fetch energy savings, we do see that the main results are relatively stable across different ratios.

8. ESTIMATION-BASED EXPLORATION

8.1 Motivation

A designer of an embedded microprocessor platform might run simulations like those above, for the particular benchmarks of interest and the particular technology being used, to pick the best tiny caching method. For our benchmarks and technology, for example, we might choose to include a one-level hybrid loop cache of size 128 instructions.
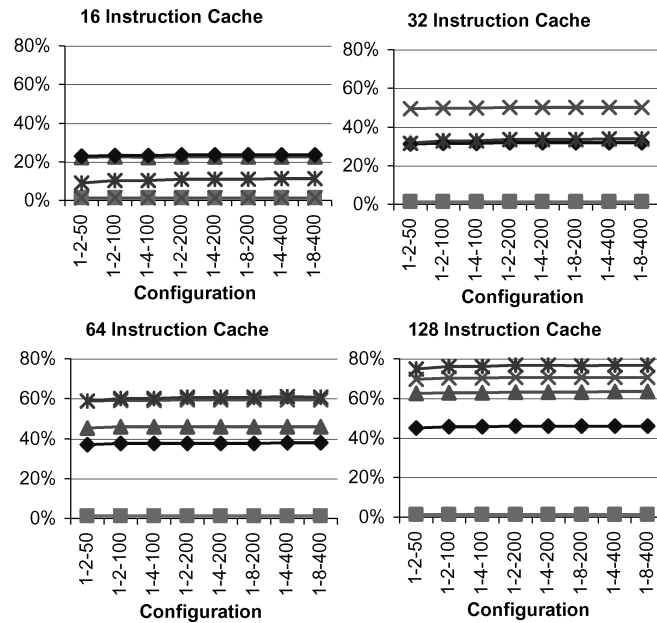
Fig. 14. Instruction access energy savings for various power ratios for MediaBench running on SimpleScalar (see Figure 11 for legend).

However, many microprocessors are now available as cores, synthesized onto part of a chip along with other cores, by an embedded system designer. This provides the opportunity to tune the tiny cache (as well as the rest of the memory hierarchy) to the particular program that will be running on the microprocessor. While the embedded system designer might be willing to setup and perform simulations like those above, a core provider would likely be better off providing a tool that quickly and automatically chooses the best tiny cache configuration for a given program.

We therefore developed a faster method than loop cache simulation for exploring the loop cache configuration space [Cotterell and Vahid 2002]. As previously mentioned, *lcsim* follows the paradigm of traditional cache simulators, like Dinero [Edler and Hill n.d.]. Thus, we could look into traditional cache simulation speedup methods, like examining multiple configurations per pass [Sugumar and Abraham 1991] or compacting the trace size using statistical methods [Wu and Wolf 1999]. However, we found that due to the nature of loop caches, a faster and simpler estimation method was possible. In short, we could apply simple equations and algorithms to the loop statistics in order to generate adequately accurate loop cache statistics.

As shown in Figure 15, in the estimation-based approach for analyzing various loop caches, the loop cache simulator from our simulation methodology was replaced with an equation-based estimator. The loop cache simulator utilizes the program instruction trace as well as the loops to be cached, which are read each time the simulator is run, resulting in long simulation times. In the estimation approach, LOOAN [Villarreal et al. n.d.] reads the program's
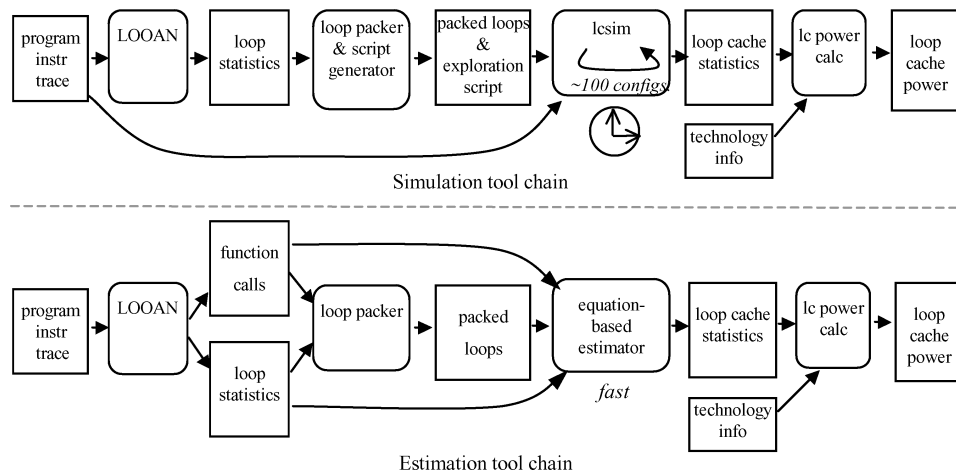
Fig. 15.   Simulation and estimation-based loop cache configuration synthesis methods.

instruction trace once and the estimation methodology uses the information generated to determine the loop cache statistics for each loop cache. Furthermore, LOOAN was modified to output the addresses at which function calls were made to create a more complete picture of the executing program. Once these data are generated, we then use the various estimation techniques described below to statically analyze each benchmark. The goal of each estimation technique is to determine the loop cache statistics—the number of instruction memory fetches, detection operations (i.e., checking to see if we should execute from the loop cache or not), number of instructions filled into loop cache, and the number of instructions fetched from the loop cache—without running the time-consuming *lcsim* at all.

To determine the number of various operations, we take the loop hierarchy provided by LOOAN and iterate through each loop. Then, for each loop, we accumulate the estimated number of fills, fetches, and detects corresponding to only the loop we are currently investigating. The estimation method varies according to the cache type being considered. We now discuss the estimation strategy for two loop cache designs: the dynamically loaded loop cache and the preloaded loop cache.

8.2 Dynamically Loaded Loop Cache

In the dynamically loaded loop cache, we are interested in the number of times we fill the loop cache with an instruction, the number of times we fetch an instruction from the loop cache, and the number of L1 fetches. Since the dynamically loaded loop cache contains no preloaded loops, there are no loop address registers we must compare addresses with, thus no detect operations.

On the first iteration of each loop, the loop cache controller sees an sbb that triggers filling the loop cache on the second iteration. The loop cache will continue to fill until a cof is detected. Thus, to estimate the number of fill instructions, we first check whether this loop would iterate at least two times, since

otherwise the loop cache would never be filled with this loop. We then want to see how many instructions from this loop would be filled into the loop cache. We determine where the first cof will occur. This cof can originate from the sbb that trigged the fill, an sbb from a subloop, or a function call. The cof may also correspond to a jump, but this information is not provided in the static analysis. If the current loop contains subloops, the loop cache controller will fill to the end of the first subloop. Similarly, if the loop contains a function call, the function call map previously generated from LOOAN is used to determine the exact instruction from which the function call originates. The smallest of the three aforementioned addresses is determined and from it we subtract the start address of the loop we are interested in. We then check to see if the start address minus the above computed address is larger than the loop cache size. If it is, we set the number of instructions filled on a given iteration to the loop cache size. This calculation is the number of instructions that will be filled into the loop cache. Therefore, each time this particular loop is called, it will fill that many instructions, so we then multiply this number by the number of times the loop is executed.

The dynamic loop cache fetches instructions starting with the third iteration of the loop. Once again, the loop cache controller will stop fetching when a cof is detected. To calculate the number of fetches, we check to see that the average number of iterations is greater than or equal to three. If not, this loop will never be fetched from the loop cache. The number of instructions fetched per iteration is determined using the same method used above to determine the number of instructions to fill. Thus, we multiply the number of instructions fetched within the loop by the iteration average minus two. Additionally, this behavior occurs every time the loop is executed, hence, we multiply the fetches per execution by the number of times the loop is called.

Finally, we fetch an instruction from L1 memory when it is not fetched from the loop cache. Using the output from LOOAN indicating the total number of instructions executed, we obtain the number of L1 fetches by subtracting from the total number of instructions executed the number of fetch operations we previously determined.

8.3 Preloaded Loop Caches

The preloaded loop cache scheme requires that we select loops beforehand via profiling. These loops are never replaced, thus the number of dynamic fills for this type of caching scheme is always zero (fills occur before regular program execution).

Since loops are preloaded, only those loops will contribute to the number of fetches. In addition, we must wait for a cof to trigger the controller to compare the address with the loop address registers to see if the loop is preloaded. Since cofs caused by jump instructions are not provided in the static analysis, there are only two cases for which we can determine when a cof occurs. The first case is when the loop executes its first iteration. The loops sbb will trigger a detect operation and on the second iteration, the corresponding instructions are fetched from the loop cache. Thus, the number of fetches contributed by

this loop is equal to the number of times the loop is executed, multiplied by the number of average iterations minus one. The second potential cof occurs when a function is called from within the loop. There is a cof to call the function and a cof to return from the function. Upon returning to the loop from the function call, the cof will trigger fetching starting from the location following the function call. Thus, in this situation the number of fetches contributed by this loop is equal to the total number of instructions executed. However, to account for the above situation, the number of executions multiplied by the difference between the function call address and the starting address of the loop is subtracted from the total for this loop.

The number of detect operations corresponds to the number of cofs in the given program. There exists a cof at the end of a loop, when calling a function, and when returning from a function. Thus for every loop we find, we add a detect operation to represent the sbb at the end of the loop. For every function call we add two detect operations.

Once we have gathered the various statistics we are interested in for each of the cache configurations considered, we feed this information into another program, which calculates the power of each loop cache design. The relative capacitance values can be varied in this stage so that the power values outputted correspond to the desired technology.

8.4 Results

In evaluating our simulation-based and estimation-based approaches, we need to analyze the results for each with respect to accuracy and fidelity between the two approaches. To determine the accuracy of the estimation method, we first ran each of the benchmarks through the loop cache simulator to obtain the power savings for each cache configuration over a configuration without a cache. Next, each benchmark was run through the loop cache estimator to obtain the power savings of each cache configuration over a configuration without a cache.

We then compared the average power savings reported for each cache configuration over all benchmarks using the simulation-based method versus the average power savings reported for each cache configuration over all benchmarks using the estimation-based method. This comparison is shown in Figure 16 (these data utilize net ratios of 1:8:100, as discussed in Section 7). Cache configurations 1 through 8 are for the dynamic loop cache and configurations 9 though 48 are for the preloaded loop cache. For the dynamic loop cache, the estimation method reported approximately 15% more power savings than reported by the simulation-based results. For the preloaded loop cache, the estimator reported—1% to 3% difference in power savings. On average, the estimation methodology had an average accuracy of 2%.

While the relative accuracy of the estimated power savings is important, in order for this approach to be viable, there must be fidelity between the choices selected under each approach as the best loop cache configuration. Therefore, to ensure any inaccuracies from estimation do not compromise the fidelity, for each benchmark we selected the loop cache configuration chosen as the best
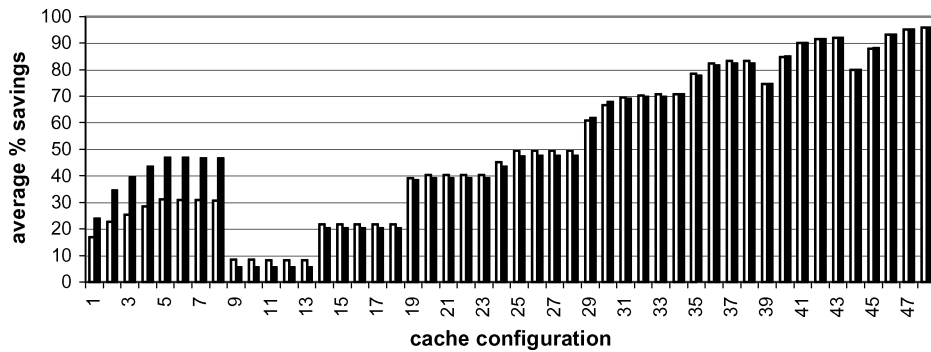
Fig. 16. Percent savings for various cache configurations, using simulation (white bars) versus using estimation (black bars) for averages over all benchmarks.
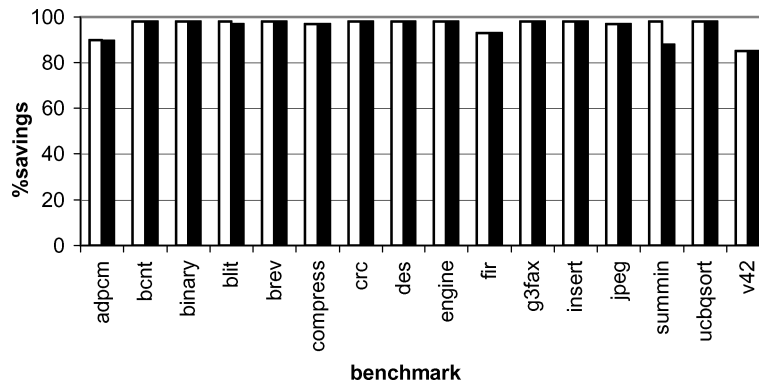

Fig. 17. Power savings using cache configurations from simulation approach (white bars) versus estimation approach (black bars).

by both the simulation-based approach and the estimation-based approach. Figure 17 shows the power savings for the cache configuration selected by the simulation-based approach versus the power savings using the cache configuration selected by the estimation-based approach across all benchmarks. In most cases, the cache configuration selected as best by the estimation method saves as much power as the cache configuration selected as best by the simulation methodology. The worst difference in performance of the loop cache obtained from estimation versus simulation is for the *summin* benchmark, where the estimation approach selects a cache configuration that is 10% less than the optimal configuration. However, on average the cache configuration obtained through estimation is less than 1% away from the optimal reported by the simulation method.

We have shown that through estimation we have good accuracy and preserve fidelity. Now we describe the speedup obtained by using estimation rather than simulation. Table VIII shows the breakdown of time spent in various areas of the simulation based approach for each benchmark. In addition, the breakdown of time spent in various areas of the estimation-based

Table VIII. Speed of Simulation Versus Estimation (in s)

| Benchmark | Number Instructions Executed | Simulation Tool Chain | | | | | Estimation Tool Chain | | | |
		lc Pwr Calc	LOOAN	Script Gen	lcsim	Total Time	LOOAN	Estimator	Total Time	Speedup
adpcm	63,891	0.01	0.31	0.01	32.15	32.48	0.31	0.16	0.48	68
bcnt	1,938	0.01	0.01	0.01	1.17	1.20	0.02	0.06	0.09	13
binary	816	0.01	0.01	0.01	0.87	0.90	0.01	0.08	0.10	9
blit	22,845	0.01	0.07	0.01	7.26	7.35	0.07	0.06	0.14	53
brev	2,377	0.01	0.01	0.01	1.20	1.23	0.01	0.06	0.08	15
compress	138,573	0.01	0.85	0.01	82.50	83.37	0.85	0.14	1.00	83
crc	37,650	0.01	0.15	0.01	16.03	16.20	0.15	0.07	0.23	70
des	122,214	0.01	0.44	0.02	45.28	45.75	0.44	0.07	0.52	88
engine	410,607	0.01	2.12	0.02	214.99	217.14	2.12	0.08	2.21	98
fir	16,211	0.01	0.07	0.02	7.60	7.70	0.07	0.09	0.17	45
g3fax	1,128,023	0.01	3.54	0.02	385.44	389.01	3.54	0.09	3.64	107
insert	1,942	0.01	0.01	0.01	1.18	1.21	0.01	0.06	0.08	15
jpeg	4,594,721	0.01	17.57	0.01	1837.28	1854.87	17.57	0.12	17.7	105
summin	1,909,787	0.01	11.42	0.01	903.73	915.17	8.25	0.09	8.35	110
ucbqsort	219,978	0.01	0.93	0.01	82.62	83.57	0.89	0.09	0.99	84
v42	2,442,551	0.01	12.07	0.01	1252.48	1264.57	12.27	0.12	12.4	102
									Average	67

approach for each benchmark is also shown. All simulations and estimations were executed on a 500 MHz Sun Ultra60 workstation. Table VIII shows that the majority of time for the simulation-based method is spent running the loop cache simulator (lcsim). Thus, by decreasing this time by using estimation, a significant speedup is achievable. For the larger examples, jpeg, summin, and v42, the simulation-based approach required approximately 30 min, 15 min, and 21 min, respectively. However, by using the estimation-based method the times required were reduced to approximately 17 s, 8 s, and 12 s, respectively. While many of the other benchmarks did not require a very long time for simulation due to their small size, the estimation approach still resulted in significant speed up. Overall, the speedup using estimation ranges from 9 to 109 across various Powerstone benchmarks, with an average speedup of 67. For the larger examples in MediaBench, we found that the simulation-based method takes tens of hours, while estimation still requires only seconds to minutes.

9. CONCLUSIONS

Adding a tiny instruction cache to the instruction memory hierarchy can significantly reduce the energy related to instruction fetching. The traditional method of using a tiny direct-mapped cache incurs significant performance overhead. Loop caches eliminate such overhead by only storing code known to be a small loop. We showed that dynamically loaded loop caches can reduce fetches to regular instruction memory by about 30%, while preloaded loop caches achieve about a 60–70% reduction, for Powerstone and MediaBench benchmarks. We found a hybrid dynamic/preloaded loop cache to work best on average. The instruction fetch reductions translate to instruction fetch energy savings of 60–70%. We showed that the results apply across a variety of technologies and architectures. We demonstrated a method to quickly determine the best loop cache size and type for a given program. Loop caches can thus be used as part of an energy saving methodology to help reduce overall system energy.

REFERENCES

AGHAGHIRI, Y., FALLAH, F., AND PEDRAM, M. 2001. Irredundant address bus encoding for low power. *International Symposium on Low Power Electronics and Design*, 82–87.

ALBONESI, D. H. 2000. Selective cache ways: On-demand cache resource allocation. *Journal of Instruction Level Parallelism*.

Artisan. http://www.artisan.com.

BAJWA, R. S., HIRAKI, M., KOJIMA, H., GORNEY, D., NITTA, K., SHRIDHAR, A., SEKI, K., AND SASAKI, K. 1997. Instruction buffering to reduce power in processors for signal processing. *IEEE Transactions on VLSI Systems*, 417–424.

BELLAS, N., HAJJ, I., POLYCHRONOPOULOS, C., AND STAMOULIS, G. 1999. Energy and performance improvements in microprocessor design using a loop cache. In *International Conference on Computer Design*, 378–383.

BENINI, L., DEMICHELI, G., MACII, E., SCIUTO, D., AND SILVANO, C. 1998. Address bus encoding techniques for system-level power optimization. In *Design Automation and Test in Europe*.

BROOKS, D., TIWARI, V., AND MARTONOSI, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *27th Annual International Symposium on Computer Architecture*.

BURGER, D., AUSTIN, T., AND BENNET, S. 1996. Evaluating future microprocessors: the simplescalar toolset. University of Wisconsin-Madison. Computer Science Department. Tech. Report CS-TR-1308.

Compaq Western Research Labs n.d. CACTI 3.0, http://research.compaq.com/wrl/people/jouppi/CACTI.html.

COTTERELL, S. AND VAHID, F. 2002. Synthesis of customized loop caches for core-based embedded systems. In *International Conference on Computer Aided Design*.

EDLER, J. AND HILL, M. D. n.d. Dinero IV trace-driven uniprocessor cache simulator, http://www.cs.wisc.edu/~markhill/DineroIV

GOVINDARAJAN, S. C., RAMASWAMY, G., AND MEHENDALE, M. 2001. Area and power reduction of embedded DSP systems using instruction compression and re-configurable encoding. In *International Conference on Computer Aided Design*.

HASEGAWA, A., KAWASAKI, I., YAMDA, K., YOSHIOKA, S., KAWASAKI, S., AND BISWAS, P. 1995. SH3 high codes density, low power. *IEEE Micro*.

The International Technology Roadmap for Semiconductors (ITRS). Semiconductor Industry Association. 1999.

ISHIHARA, Y. AND YASUURA, H. 2000. A power reduction technique with object code merging for application specific embedded processors. In *Design Automation and Test in Europe*.

JOUPPI, N. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *International Symposium on Computer Architecture*, 364–373.

KIN, J., GUPTA, M., AND MANGIONE-SMITH, W. 1997. The filter cache: An energy efficient memory structure. In *International Symposium on Microarchitecture*, 184–193.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communication systems. In *Proceedings 30th Annual International Symposium on Microarchitecture*.

LEE, L. H., MOYER, B., AND ARENDS. 1999a. Low-cost embedded program loop caching—revisited. U. Mich. Technical Report Number CSE-TR-411-99.

LEE, L. H., MOYER, B., AND ARENDS, J. 1999b. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *International Symposium On Low Power Electronics and Design*.

MALIK, A., MOYER, B., AND CERMAK, D. 2000. A low power unified cache architecture providing power and performance flexibility. In *International Symposium on Low Power Electronics and Design*.

MOYER, B., LEE, L. H., AND ARENDS, J. 1999. Data processing system having a cache and method thereof, US Patent number 5, 893, 142.

SEGARS, S. 2001. Low power design techniques for microprocessors. In *IEEE International Solid-State Circuits Conference*.

SIAS, J. W., HUNTER, H. C., AND HWU, W. W. 2001. Enhancing loop buffering of media and telecommunications applications using low-overhead predication, In *Proc. of the 34th International Symposium on Microarchitecture*.

STAN, M. R. AND BURLESON, W. P. 1995. Bus-invert coding for low-power I/O. *IEEE Transactions on Very Large Scale Integration Systems 3*, 1, 49–58.

SUGUMAR, R. AND ABRAHAM, S. 1991. Efficient simulation of multiple cache configurations using binomial trees. Technical Report CSE-TR-111-91, CSE Division, University of Michigan.

Synopsys Inc. n.d. http://www.synopsys.com.

VILLARREAL, J., LYSECKY, R., COTTERELL, S., AND VAHID, F. n.d. Loop analysis of embedded applications. UC Riverside CS&E Technical Report UCR-CSE-01-03.

VILLARREAL, J., SURESH, D., STITT, G., VAHID, F., AND NAJJAR, W. 2002. Improving software performance with configurable logic. *Design Automation of Embedded Systems*.

WU, Z. AND WOLF, W. 1999. Iterative cache simulation of embedded CPUs with trace stripping. In *International Conference on Hardware/Software Co-Design*.