

Cool-Cache: A Compiler-Enabled Energy Efficient Data Caching Framework for Embedded/Multimedia Processors

OSMAN S. UNSAL, RAKSIT ASHOK, ISRAEL KOREN, C. MANI KRISHNA,
and CSABA ANDRAS MORITZ

University of Massachusetts, Amherst

The unique characteristics of multimedia/embedded applications dictate media-sensitive architectural and compiler approaches to reduce the power consumption of the data cache. Our goal is exploring energy savings for embedded/multimedia workloads without sacrificing performance. Here, we present two complementary media-sensitive energy-saving techniques that leverage static information. While our first technique is applicable to existing architectures, in our second technique we adopt a more radical approach and propose a new tagless caching architecture by reevaluating the architecture–compiler interface.

Our experiments show that substantial energy savings are possible in the data cache. Across a wide range of cache and architectural configurations, we obtain up to 77% energy savings, while the performance varies from 14% improvement to 4% degradation depending on the application.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*; C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Low-power design, compiler-architecture interaction, cache partitioning, tagless caching

1. INTRODUCTION

The recently introduced low-power media/embedded processors share a common trait; the caches consume a significant portion of the power consumed: 42% and 23% of the total processor power in StrongARM 110 [Montenaro et al. 1996] and Power PC [Bechade et al. 1994], respectively (see Figure 1). Therefore, saving cache energy will have a considerable impact on the overall energy consumption.

In an earlier paper about the FlexCache project [Moritz et al. 2001], we described our vision of a multipartitioned cache where memory accesses are

Parts of this work were presented at the WMP1'01 [Unsal et al. 2001a], MICRO-34 [Unsal et al. 2001b], and HPCA-8 [Unsal et al. 2002].

Supported in part by NSF grants EIA-0102696, CCR-0205212 and CCR-0105516.

Authors' address: Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA, 01003.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 2003 ACM 1539-9087/03/0800-0373 \$5.00

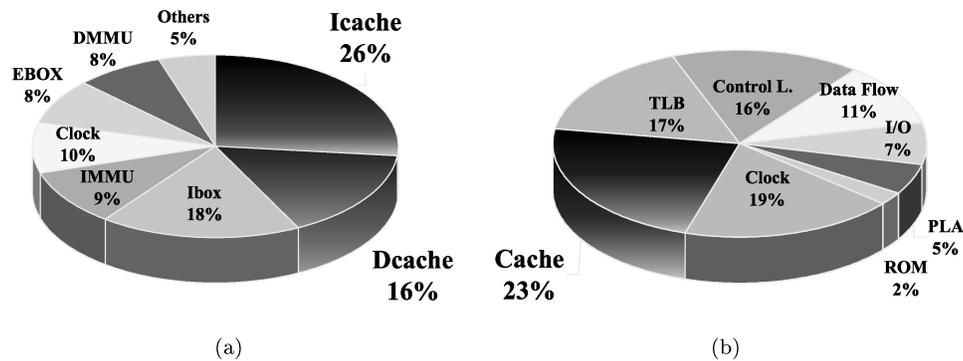


Fig. 1. Power consumption for embedded/media processors. (a) StrongARM; (b) PowerPC.

separated based on their static predictability and memory footprint, and managed with various compiler-controlled techniques supported by instruction set architecture extensions, or with traditional hardware control. Here we apply our vision to data cache energy savings. To implement this goal, we blur the boundary between the architecture and compiler layers.

In particular, our contributions are

- A compiler-controlled data remapping scheme directs scalar accesses to a small scratchpad SRAM area. This scheme can be utilized in existing media processors and results in up to 38.2% average energy savings without sacrificing performance.
- Hotlines*: An embedded/media-sensitive compiler-enabled caching framework that eliminates cache tags. *Hotlines* is up to 50% more energy efficient than a regular cache.

We adopt an incremental approach. In the first phase, we employ data partitioning for scalars. This approach requires few, if any, modifications to current architectures and compilers. We examined the memory footprint of scalars in embedded/multimedia applications and found them to be extremely small Unsal et al. [2001]. We also established that a significant percentage of memory accesses in these applications are scalar accesses. These characteristics motivated us to direct the scalar accesses to a small scratchpad SRAM area. Although accessed very frequently, this small SRAM is more energy efficient than when scalar data are mapped into the large L1 cache.

In the second phase, we aim for greater energy savings through graceful but powerful architectural/compiler paradigm redefinitions. We design and introduce a compiler-controlled tagless caching framework, *hotlines*, which achieves significant energy savings. Our *hotlines* framework saves energy without substantial performance loss, in some cases even beating traditional hardware-based cache performance. The compiler-directed cache is a flexible, compiler-generated data cache that replaces the tag-memory and cache controller hardware with a compiler-managed tag-like data structure. Being software based, the cache is highly reconfigurable—such parameters as line-size and associativity can be tailored to each application to provide maximum performance.

The virtual address to SRAM address translation, conventionally performed by the cache controller, must now be done by the software. Typical steps would involve extraction of tag, set, and line-offset bits from the address, comparing the tags (the tag-structure is itself stored in the SRAM), and so on. This will take several cycles compared to just one for the hardware cache. Fortunately, there is considerable reuse of such address translation and the compiler can be modified to take advantage of it. Most of the memory accesses for any application in general, and media applications in particular, are generated by array accesses which have a high degree of locality. Consider an array $A[i]$ being accessed in a loop. If A has 8-byte-wide elements and the cache line is 256 bytes wide, we have 32 elements per cache-line. Now if $A[i]$ is being accessed sequentially, we will have one new address translation followed by 31 reuses of this translation. This is where the hotline technique becomes beneficial. We propose an 8-entry hotline register file that caches eight virtual address to SRAM address translations. The hotline compiler pass assigns each nonscalar (arrays, structures, and so on) a unique hotline register. Every time a nonscalar is accessed; the emitted virtual address is compared with virtual address contained in the hotline register associated with this nonscalar. If it matches, we have a hit (address translation reuse). On a miss, a software exception handler is invoked to do the translation and update the hotline register with the new translation. Since this reuse scheme is generated by the compiler, we call it static prediction. For the applications tested, the static prediction rate is found to average around 80%.

This paper is organized as follows. In Section 2 we review related work and reiterate our motivation. In Section 3 we present the architectural framework for our incremental techniques. We address compiler issues in Section 4. Section 5 explains our experimental setup. We divide our results section into two: in Section 6.1 we analyze the energy efficiency of our scratchpad technique in isolation. We then embed this technique in our proposed Cool-Cache framework together with our *hotlines* approach and study the performance and energy savings of the complete Cool-Cache framework in Section 6.2. We conclude in Section 7.

2. PREVIOUS WORK

Previous cache-partitioning research focused more on performance issues rather than on energy. Providing architectural support to improve memory behavior includes split caches that were discussed in Milutinovich et al. [1996]. Albonesi [2000] proposed selective cache ways, a vertical cache-partitioning scheme. Benini et al. [2000] discuss an optimal SRAM partitioning scheme for an embedded system-on-a-chip. Panda et al. [1997] propose use of a scratchpad memory in embedded processor applications. Kin et al. [1997] study a small L0 cache that saves energy while reducing performance by 21%. Lee and Tyson [2000] use the mediabench benchmarks and have a coarse-granularity partitioning scheme: they opt for dividing the cache along OS regions for energy reduction. Chiou et al. [2000] employ a software-controlled cache and use a cache-way-based partitioning scheme. A recent paper by Huang et al. [2001] also presents a way-prediction scheme; their cache partitioning

includes a specialized stack cache, and compiler implementation concerns are addressed.

Combined compiler/architectural efforts toward increasing cache locality [Memik et al. 1999] have exclusively focused on arrays. A recent memory behavior study for multimedia applications has also primarily targeted array structures [Kulkarni et al. 2000]. Another recent paper by Delaluz et al. [2000] discusses energy-directed compiler optimizations for array data structures on partitioned memory architectures; they use the SUIF compiler framework for their analysis. One previous work that also targeted multimedia systems [Ranganathan et al. 2000] has considered dynamically dividing caches into multiple partitions, using the Mediabench benchmark in the performance analysis, with comments on compiler-controlled memory. Cooper and Harvey [1998] look at compiler-controlled memory. Their analysis includes spill memory requirements for some Spec '89 and Spec '95 applications. Witchel et al. [2001] propose a direct-addressed cache that eliminates some cache tag accesses and thereby saves energy. In their study of instruction fetch prediction, Calder and Grunwald [1995] introduce a tagless memory buffer for next cache line and set prediction. Abraham and Mahlke [1999] evaluate memory hierarchies for embedded systems from a performance point of view.

Our previous work [Moritz et al. 1999, 2001; Unsal et al. 2000a, 2000b, 2001a, 2001b, 2002] and the above research provide the framework and the motivation for this study. Our unique contribution is the design of an energy-efficient compiler-controlled dynamically configurable tagless caching framework. This work pushes caching further up to the compiler layer.

3. ARCHITECTURAL FRAMEWORK

3.1 Scalar Data Remapping

Our first energy-saving technique remaps every scalar memory access into a scratchpad memory area. It can be used in existing architectures, and no architectural modifications are necessary since many media/embedded processors have a scratchpad. For example, any entry in the cache in Fujitsu Sparclite can be locked, in effect making the entry an element in the SRAM buffer. Part of the cache can be reorganized as a SRAM scratchpad area in the Samsung ARM7 and Hitachi SH2. The recently introduced Intel StrongARM SA-1110 [Intel 2000] has a 512 byte minicache for frequently used data. In our previous study of the Mediabench benchmarks [Unsal et al. 2001b], we found that a slightly larger scratchpad SRAM size of 1024 bytes is enough to map all the scalars. A scratchpad SRAM guarantees single-cycle access time to scalars since there are no cache misses. Thus, we guarantee *at least* the same level of performance from our scheme as compared to a regular nonpartitioned architecture. In fact, since we decrease the cache interference, we get better data cache performance by separating scalar accesses from array accesses [Unsal et al. 2001b]. If the embedded/multimedia processor is not equipped with any kind of scratchpad mechanism, then the ISA can be augmented with special load/store instructions that would channel the scalar data to a separate cache area.

The implementation is simple: encode a single additional bit in the instruction, thus “marking” the load/store to be diverted. This is similar to the approach taken by Calder and Grunwald [1994] for marking branch instructions.

3.2 Cool-Cache Architecture

Our caching architecture is completely compiler managed, and is therefore able to leverage static information that is lost in traditional hardware caches.

The Cool-Cache architecture combines four cache control techniques: (1) fully static, (2) statically speculative, (3) hardware-supported dynamic, and (4) software-supported dynamic.

The fully static cache management is based on a disambiguation between the scalar and nonscalar accesses. As described in Unsal et al. [2001b], although the scalars typically have a very small footprint, they are frequently accessed, and have considerable interference with nonscalar accesses. The Cool-Cache architecture, by statically diverting the scalar and nonscalar accesses to the scratchpad memory and the SRAM, respectively, not only eliminates this interference but also saves power by only accessing a small scratchpad memory instead of a much larger data-array. Although our current implementation is based on statically mapping scalars, a generalization of this idea is to map frequently accessed memory references that have a small footprint into the scratchpad area.

The second technique in the Cool-Cache architecture is based on a compile-time speculative approach to eliminate tag-lookup for nonscalar memory accesses. In addition, some of the cache logic found in associative caches can also be eliminated. The idea is that if a large percentage of cache accesses can be predicted statically, then we can eliminate the tag-array and the cache logic found in associative caches and thus reduce power consumption.

The scalars are directly mapped to the scratchpad memory; no runtime overhead is required. However, if managed explicitly in the compiler, the nonscalars require virtual-to-SRAM address mappings or translations at runtime. This mapping is basically a translation of virtual cache line addresses into SRAM lines, based on the line sizes assumed in the compiler. Note that the partitioning of the SRAM into lines is only logical: the SRAM is mainly accessed at the word level, except during fills associated with cache misses. This translation can be done by inserting a sequence of compiler-generated instructions, at the expense of some software overhead. But as discussed in Moritz et al. [1999], for many applications, there is substantial reuse of these address mappings. Our findings for multimedia applications also confirm this. The compiler can speculatively register-promote the most recent translations into a small new register area: we call it the *hotline register file*. With special memory instructions, similar to those proposed in the FlexCache architecture [Moritz et al. 2001], the runtime overhead of speculation checking can be completely eliminated.

The third technique helps to avoid paying the high penalty of a software-based recovery mechanism, (i.e., during a statically mispredicted access) we use a small 16-entry fully associative cache TLB to cache address mappings for memory accesses that are mispredicted. We found that a 16-entry cache TLB

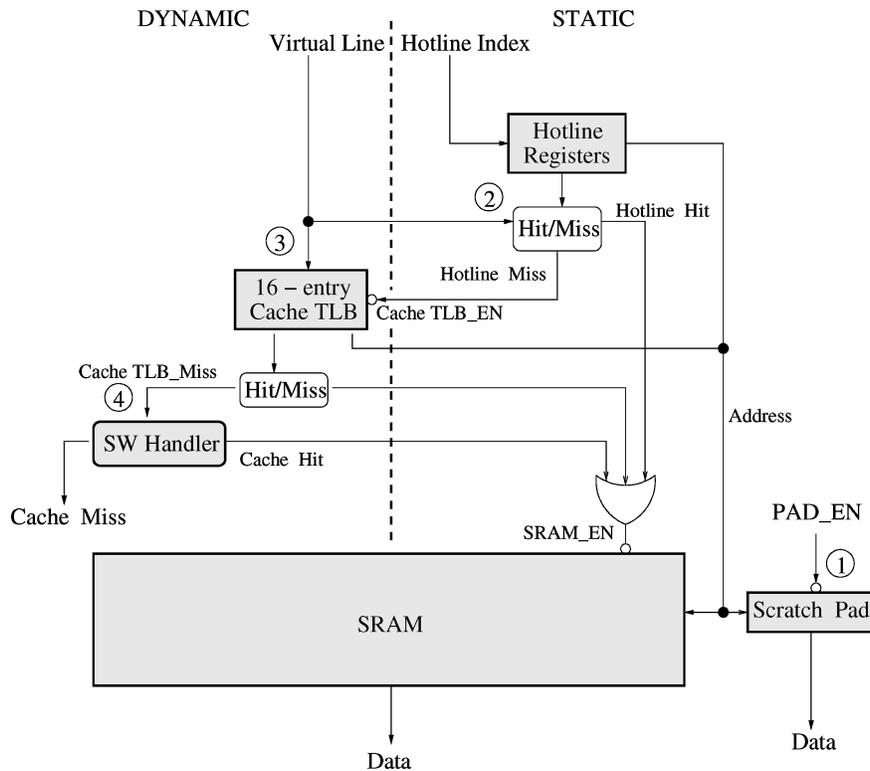


Fig. 2. The Cool-Cache architecture.

is enough to catch most of the address translations that are not correctly predicted statically. This approach is similar to caching frequently used page table entries in the TLB and to minimize address translation overhead in virtual memory systems. Further, because the hotline check can be performed at an early pipeline stage, and is very quick, we can access the cache TLB on hotline mispredictions without any performance penalty.

The fourth technique used in Cool-Cache is basically a fully reconfigurable software cache. This technique is more of a backup solution, and it can implement a highly associative mapping. Our implementation is based on a four-way associative cache with random replacement. The mapping table between virtual cache lines and physical SRAM lines is implemented similarly to an inverted page table. We have assumed a 25-cycle overhead associated with this software backup mechanism (in addition to any further cache miss latencies). Our results show that the combined static and *cache TLB* techniques capture more than 99% of the memory accesses for most of the multimedia applications.

Figure 2 provides an overview of the Cool-Cache architecture. All the memory accesses are diverted by the compiler to either the scratchpad or the hotline architecture. The scratchpad access mechanism consumes very little power due to its small size (we assume a 1 KB structure in our experiments) compared to the regular SRAM data array.

The nonscalar memory instructions carry a hotline index. This identifies the hotline register, predicted by the compiler to contain the address translation for the current memory access. Using this index, the corresponding hotline register is read from the hotline register file.

The hotline register contains the virtual cache line address to SRAM line address mapping. If the memory reference has the same virtual address as that contained in the hotline register, we have a correct static prediction. Upon a correct static prediction, the SRAM can be accessed through the SRAM address contained in the hotline register that is combined with the offset part of the address, and the memory access is satisfied. If we have a static misprediction, though, the *cache TLB* is checked for the translation information.

If the *cache TLB* hits, the hotline register is updated with the new translation, and the memory access is satisfied. A *cache TLB* miss invokes a compiler-generated software handler. This handler checks the tag-directory (which is itself stored in a nonmapped portion of the SRAM) to check if it is a cache hit or miss. On a miss, a line is selected for replacement and the required line is brought into its place, the replacement being handled by software. The cache TLB and the hotline register are updated with the new translation, and the memory access is satisfied by accessing the SRAM.

Because the software handler is accessed so seldom, its overhead has minimal effect on the overall performance. The Cool-Cache can, in fact, even surpass a regular hardware cache in terms of performance. For one thing, the interference between scalar and nonscalar accesses has been eliminated, resulting in higher hit-rate and better cache utilization. Second, a high associativity is emulated, without the disadvantage of the added access latency in regular associative caches. Since the SRAM access mechanism is much less complicated than a regular tagged hardware cache, there is a possibility of reduction in cycle time. As shown in Wilton and Jouppi [1996], the tag-access is on the critical path and can add as much as 30% to the access time of associative caches. Consequently, many designs either place the tag-access on a separate pipeline stage or try to balance the latency between the data-array path and the tag-path [Wilton and Jouppi 1996]. Finally, an optimal line size can be chosen on a per-application basis. From a power perspective, the Cool-Cache has substantial gains compared to a hardware cache for two reasons. First, there are no tag-lookups on scalar accesses and correctly predicted nonscalar accesses. Second, the SRAM is used as a simple addressable memory—the complicated access mechanisms of a regular cache consume more power.

Our results (in Section 6.2.1) show that, except for one application, the hotline prediction system performs better for higher line sizes. Specifically, a line-size of 1024 bytes gives the best result, among the tested line-sizes, for most of the applications. Such a big line-size can be an issue, however, when interfacing with higher level caches or the DRAM. Filling a 1K-wide cache line on a miss can take a large number of cycles. The problem of supporting different line-sizes, and especially the larger ones, can be mitigated to quite an extent by having an interleaved structure of DRAM banks and a slightly wider bus between the SRAM and the external memory. See Delaluz et al. [2000] for a discussion of energy-conscious interleaved memories.

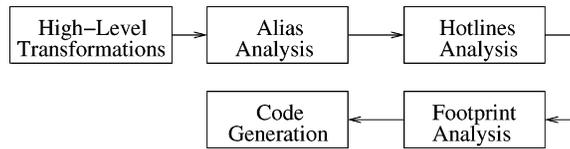


Fig. 3. Cool-Cache compiler stages.

4. COOL-CACHE COMPILER

The overall complexity of the Cool-Cache compiler is not much greater than that of a regular compiler. Figure 3 shows a high-level picture of the stages involved. The sources are first converted to the intermediate format and high-level optimizations are performed. This, the most time-consuming task, is common to both the Cool-Cache and a regular compiler. Following that is the alias analysis stage. It enables the hotline analysis to more economically assign hotlines to references. Without the alias analysis, we would liberally assign each memory reference a new hotline number. This will have a degrading effect only if the number of references within inner loop bodies is more than the number of hotlines, resulting in the same hotlines being assigned to references that could be spatially far apart. This would cause interference and result in lower prediction rates. For many applications, this does not happen, and we can omit the alias analysis stage altogether without any noticeable effect on the prediction rates.

Next we have the hotline analysis stage: this is a greatly simplified version of the algorithm used in Moritz et al. [2001], because alias analysis information is disregarded. The pseudocode of the algorithm is shown in Figure 4. The scalar footprint analysis [Unsal et al. 2001b] then calculates the footprint requirements of scalars. Having done with all the higher-level stages, code generation is performed next. This stage is modified from a regular compiler to generate the modified memory instructions: these contain the scratchpad/hotline annotations. In terms of the final binary output, the only changes we have are the additional bits in memory instructions that carry the annotations. This means that the binary can even be run on a regular hardware cache architecture that disregards the annotations. The code is exactly the same size, differing only in addresses.

5. METHODOLOGY

Since our target application is multimedia, we use Mediabench [Lee et al. 1997] in our experiments. See Table I for a short description of the benchmarks included in our analysis.

Figure 5 shows a block diagram of our framework. We needed a detailed compiler framework that would give us sufficient feedback, is easy to understand, and allows us to change the source code for our purposes. With this in mind, we chose the SUIF/Machsuiif suite as our compiler framework. SUIF performs high-level passes, while Machsuiif makes machine-specific optimizations. We have modified SUIF/Machsuiif passes for our memory remapping schemes and

```

/* For each routine, start the annotation process by starting on the first block */
for each routine do
  E = entry basic block;
  Hotline Annotate E;
end for
/* procedure to Hotline Annotate a block X */
for each non-scalar access through variable name V do
  if !(V.hasHotline) then
    increment current_Hotline; /* current_Hotline is a global variable */
    if (current_Hotline) > 8 then
      current_Hotline=1;
    end if
    V.Hotline = current_Hotline;
    V.hasHotline = true;
  end if
  annotate this memory reference with V.Hotline;
  workList = successors of X;
  while !empty(workList) do
    B = next basic block in workList;
    if B.annotated then
      continue;
    end if
    /* Traverse through the CFG by making recursive calls */
    Hotline Annotate B;
    B.annotated = True;
  end while
end for

```

Fig. 4. The hotline algorithm.

Table I. Applicable Mediabench Benchmarks

Benchmark	Description
ADPCM	Adaptive differential pulse code modification audio coding
EPIC	Image compression coder based on wavelet decomposition
G721	Voice compression coder based on G.711, G.721, and G.723 standards
GSM	Rate speech transcoding coder based on the European GSM standard
JPEG	A lossy image compression decoder
MESA	Open GL graphics clone: using Mipmap quadrilateral texture mapping
MPEG	Lossy motion video compression decoder
PEGWIT	Public key encryption coder generates a public key from a private key
RASTA	Speech recognition application

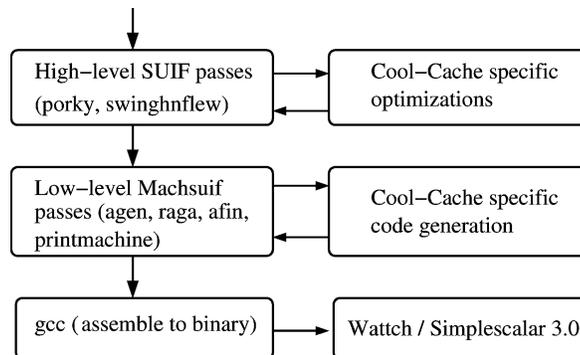


Fig. 5. Experimental setup block diagram.

Table II. Cool-Cache Specific Hardware Power Consumption. Note That L1 Data Memory is Tagless

Hardware Block Name	Modeled as	Power (W)
64 bit wide L1 memory	SRAM	6.86
256 bit wide L1 memory	SRAM	5.58
1 Kb scratchpad memory	SRAM	0.68
8 hotline registers	Register file	0.16
16 entry Cache TLB	CAM	0.34

used the SUIF annotation mechanism to propagate them. First, all the source files are converted into SUIF format and merged into one SUIF file. Then, the hotline pass (which is a SUIF pass) is run on this merged file to produce a modified SUIF file. The hotline pass analyzes the file and annotates the nonscalar accesses with hotline numbers. Next, we run this SUIF file through the Machsuif passes. The Machsuif Raga pass annotates all the scalar accesses as such. The resulting assembler code targets the Alpha processor and contains two kinds of annotations that are of interest to us: hotline and scalar annotations. We amended the assembler code by inserting NOP-like instructions around the annotated memory operations, thus *marking* them.

We then used the Wattch [Brooks et al. 2000] tool suite to run the binaries and collect the energy results. Wattch is based on the SimpleScalar [Burger and Austin 1997] framework. The simulators have been modified to recognize the annotations in the *marked* code, do hotline register checks, cache TLB checks, and so on. Such statistics as the number and energy of scalar and hotline accesses, correct static predictions, cache TLB hits, scratchpad cache, and Cool-Cache tagless SRAM accesses are output by the simulators.

Our baseline machine model is an ARM-like single-issue in-order processor. Lee and Tyson [2000] use an identical configuration in their power dissipation analysis of region-based caches for embedded processors. We modified Wattch to calculate the energy consumption of the additional hardware blocks required in Cool-Cache. The added blocks and their power consumption as modeled by Wattch are shown in Table II. Wattch uses an analytical cache energy dissipation model, similar to Kamble and Ghose [1997]. The added blocks are modeled as a SRAM (for the tagless cache), a register file (for the hotline registers), and a CAM (for the cache TLB). We use the activity-sensitive conditional clocking power model in Wattch, that is, the cache consumes power only when it is accessed. This is the model that gives the most conservative energy savings. Note that other Wattch power models reported even higher savings for our framework. An example is the *cc3* model that includes leakage power; by eliminating the separate tag-structure and logic (tags are seamlessly stored along with data in the SRAM), we save significantly on the static/dynamic power associated with the tags. To determine the baseline cache size, we did a survey of data cache sizes of current multimedia processors. As Table III indicates, the trend is towards larger caches. Therefore, we have selected a 64KB 2-way cache as our baseline. We also examine 32K and 128K caches in our sensitivity analysis. See Table IV for our baseline configuration.

Table III. Data Cache Sizes for Typical Media Processors

Processor	L1 Size	L2 Size
ARM ARM10	32K	None
Transmeta Crusoe TM3200	32K	None
Transmeta Crusoe TM5400	64K	256K
Intel StrongARM SA-110	16K	None
Equator Map-CA	32K	None

Table IV. Baseline Parameters

Processor speed	1 GHz
Process parameters	0.35 μm , 2.5 V
Issue	In-order single-issue
L1 D-cache	64 Kb, 2-way associative
L1 I-cache	32 Kb, 2-way associative
Scratchpad	1 Kb
On-chip L2 cache	None
L1 D-cache hit time	2 cycles
Scratchpad hit time	1 cycle
L2 cache hit time	20 cycles
Main memory hit time	100 cycles

5.1 Scalar Data Remapping

Our main focus is Machsuiif's register allocator pass, Raga. Raga uses a graph-coloring heuristic to assign registers to temporaries. We have made modifications to Raga to annotate scalar memory accesses. The scalar memory accesses consist of spills and register promotion-related memory accesses. Obviously, this could only be done if the memory footprint of the scalars is smaller than the scratchpad area. We presented the compiler algorithm that extracts the footprint size in Unsal et al. [2001b].

6. RESULTS

6.1 Scratchpad Energy Savings

Unless otherwise stated, all the results in this section are for a scratchpad of size 1024 bytes and a 64 KB 2-way associative baseline cache. We ran the benchmarks using the modified Wattch/SimpleScalar and collected the data cache energy results. Figure 6 shows the percentage energy savings for our 32 general-purpose register media processor model. Compared to the baseline monolithic cache, we save 10.7% energy on average by using our scheme.

Many media processors such as the ARM have a smaller number of registers, usually 16. Therefore, we have repeated our energy analysis for a 16-register version of our media processor. For 16 registers we have significantly more scalar memory accesses due to register pressure. The results are also shown in Figure 6. Our technique saves in this case an average of 38.2% of energy.

In fact, we show that we can be just as energy efficient with a 16-register media processor with a scratchpad SRAM as a 32-register processor with no scratchpad, see Figure 7. Actually, the overall energy savings are even greater

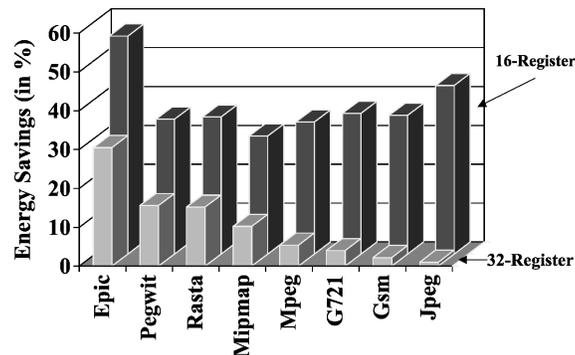


Fig. 6. Scratchpad energy savings.

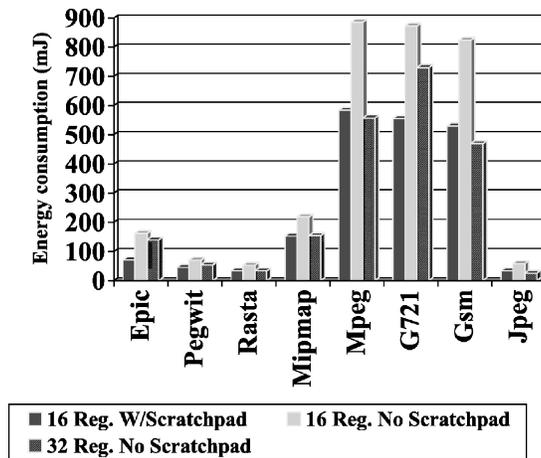
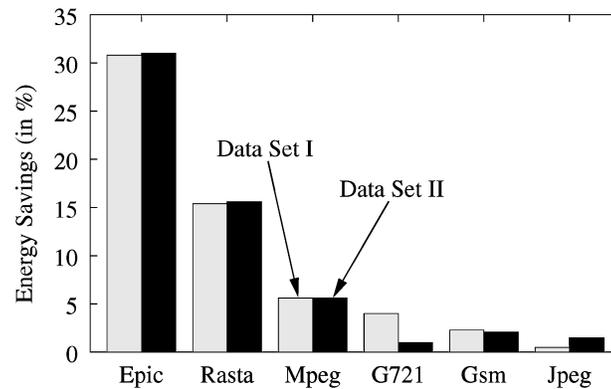


Fig. 7. 16-register architecture with scratchpad can be more energy efficient than 32-register architecture without scratchpad.

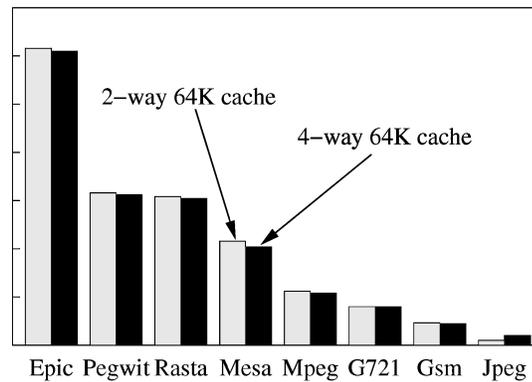
since we just concentrate on the data cache energy consumption: a 16-register file consumes substantially less power than a 32-register file.

Mediabench supplies two input sets: the second input set is larger and therefore exercises the caches more. We used this alternative input set and ran the applications for a study of the sensitivity of the energy savings to the input data set. Although the data cache energy consumption of the second set is greater, the results in Figure 8(a) suggest that the energy savings are independent of the input sets included in the Mediabench.

Next we explore the sensitivity of the energy savings to the cache associativity and size. We compared our baseline cache with a 64K 4-way cache. The results in Figure 8(b) show that the savings are fairly independent of cache associativity. We have also looked at the impact of cache size. Figure 9(a) shows the energy consumption in millijoules for three cache sizes. Figure 9(b) shows the corresponding energy savings for these cache sizes. Although the energy consumption differs according to cache size, the energy savings due to our method remain almost independent of the size.



(a)



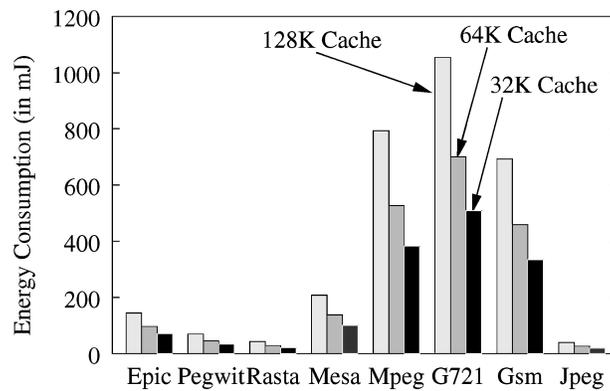
(b)

Fig. 8. Energy savings sensitivity analysis. (a) Sensitivity to input; (b) sensitivity to cache associativity.

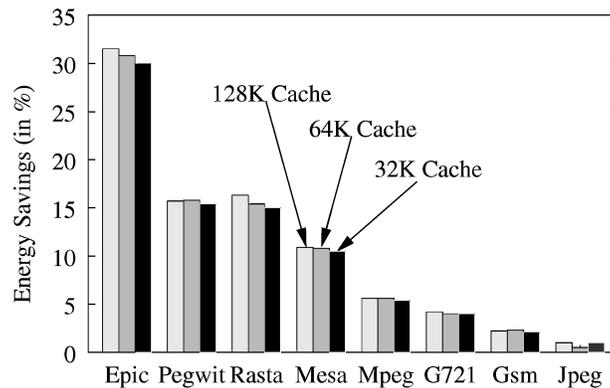
6.2 Cool-Cache Performance and Energy Savings

6.2.1 Prediction Rates. The prediction rates of the hotlines scheme are shown in Figure 10. The sensitivity of the prediction rates to both cache line-size and cache size are also shown. Figure 10(a) shows the hit rate variation as a function of cache size where the line-size has been fixed at 256 bytes. The three bars for each application, starting from the left, are for cache sizes of 32 Kb, 64 Kb, and 128 Kb, respectively. Figure 10(b) has the cache size fixed at 32 Kb with line sizes of 1024b, 256b, and 64b.

From the second graph in Figure 10(b), it can be concluded that the prediction rates (both static and dynamic) increase as the line-size increases. There are two reasons for this. First, since the media applications exhibit high-spatial locality, even a line-size as large as 1 Kb does not degrade cache performance (except for *pegwit*, where the cache-miss rate is seen to drop as line-size decreases).



(a)

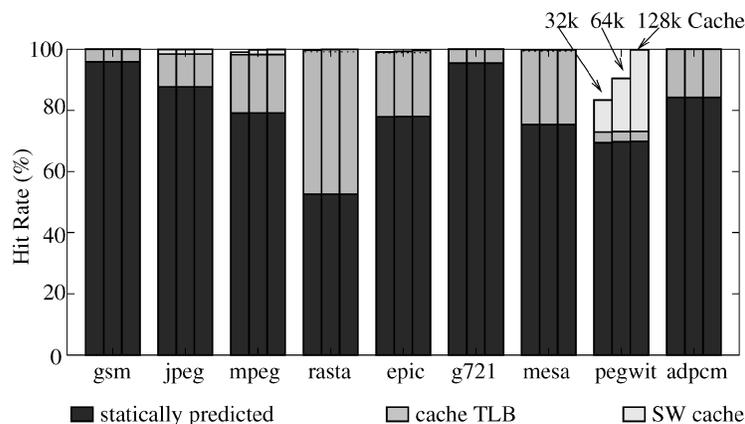


(b)

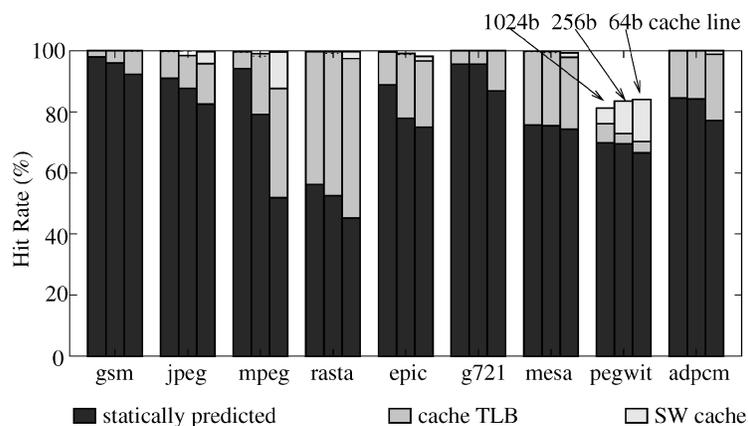
Fig. 9. Energy consumption and savings for different cache sizes. (a) Energy consumption; (b) energy savings.

Second, as the line-size increases, the memory area covered by each hotline becomes larger and there is a higher chance of correct static prediction.

Figure 10(a) shows that the static and dynamic prediction rates are almost independent of cache size. The prediction rate depends on the rate of reuse of cache lines. A high prediction rate implies that 24 cache lines (8 of which are in the hotline register file and 16 in the *cache TLB*) are being heavily reused. As long as these 24 lines are not replaced from the cache during this period of heavy reuse, the prediction rate will stay the same, regardless of the cache size. For a 256-byte line, this translates to 6 KB. Therefore, for cache sizes above, say 8 Kb, the prediction rate will be fairly constant. Figure 11 shows the sensitivity to the input data size, where the cache and line sizes have been fixed at 32 Kb and 1024 bytes, respectively. The first bar corresponds to a small input data. The second one is for a much bigger data set. As can be seen, the rates are fairly independent of the data size.



(a)



(b)

Fig. 10. Hit rate for different configurations. (a) 256 byte line; (b) 32K cache.

6.2.2 Performance. We now study the impact of the Cool-Cache on performance. Figure 12 shows the memory performance, that is, the cycles spent on memory instructions. The values have been normalized with 1 representing the hardware cache performance. There are three stacked bars for each application—for a 32K Cool-Cache with line-sizes of 1024b, 256b, and 64b, respectively. Each bar has several components. Starting from the bottom, they are time spent on scratchpad accesses, correctly predicted hotline accesses, hotline mispredictions that hit the *cache TLB*, TLB mispredictions that hit the cache, and cache misses. Since the Cool-Cache is reconfigurable, the line that gives the best performance can be chosen. The worst performer is *pegwit*, for which the memory instructions take double the time taken on a hardware cache. Note, though, that these performance numbers

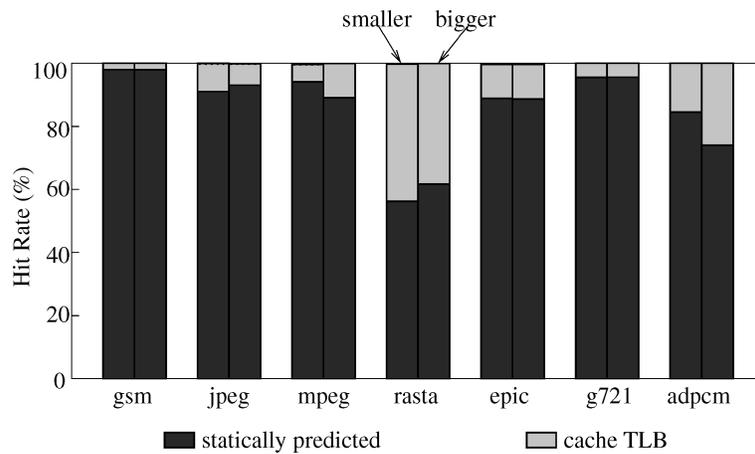


Fig. 11. Hit rate sensitivity to benchmark input.

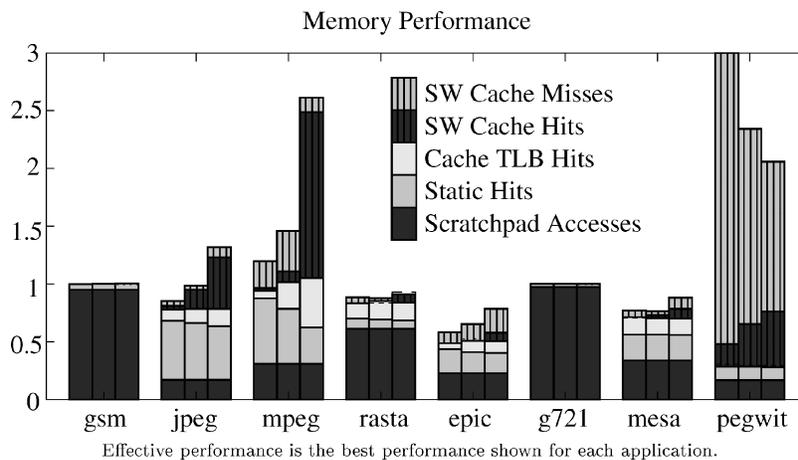


Fig. 12. Cool-Cache memory performance.

are based on cycle counts and not time: the simpler access mechanism in Cool-Cache as compared to a hardware cache can lead to a shorter cycle time.

Figure 13 shows the overall performance values (again normalized to a 0–1 scale) for the same cache size and line-size parameters. Since the memory instructions are a fraction of the total executed instructions, the overall performance boost/degradation is less than the memory performance boost/degradation. Note that for four of the benchmarks, we perform better than a hardware cache. Two benchmarks have the same performance, and for two benchmarks we have worse performance. However, the worst performance degradation is 4%; while the best performance gain, for *Epic*, is 14%.

6.2.3 Energy Savings. We now evaluate the energy savings of our Cool-Cache framework over traditional hardware caching. As in Section 6.1, this

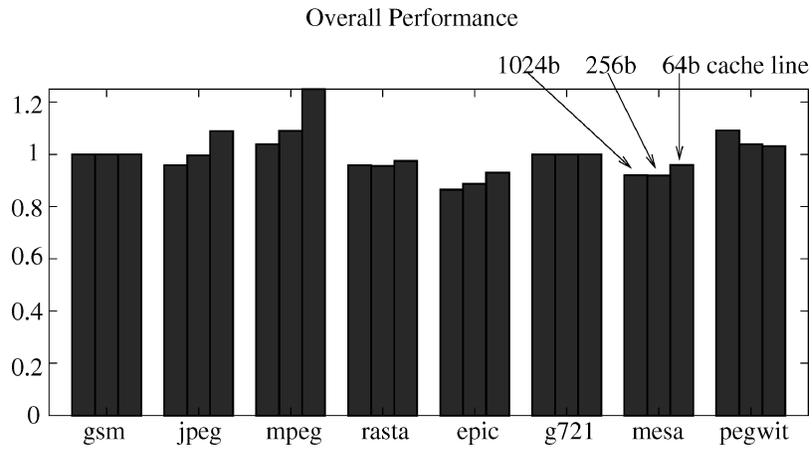
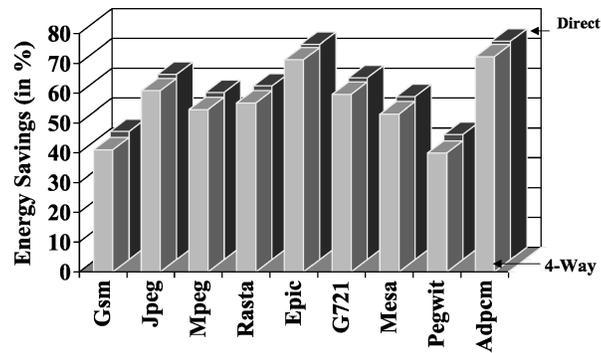
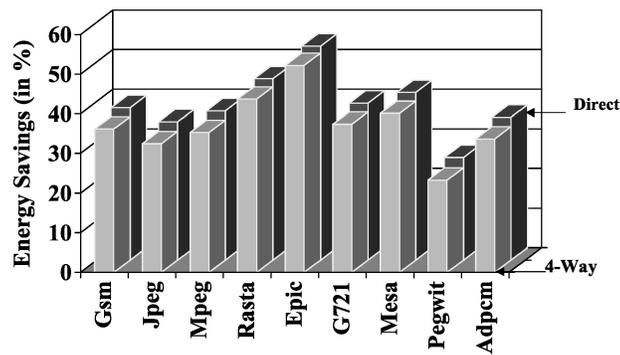


Fig. 13. Cool-Cache performance.



(a)



(b)

Fig. 14. Cool-Cache energy savings. (a) 8-byte and 32-byte wide SRAM Cool-Cache 16-register CPU; (b) 8-byte and 32-byte wide SRAM Cool-Cache 32-register CPU.

analysis is performed for two different media processor configurations: a 16-register and a 32-register CPU. Note that the actual energy savings from Cool-Cache is even greater: unlike a traditional set associative hardware cache, Cool-Cache does not need set selection multiplexers. We do not account for the energy impact of eliminating this hardware block, since Wattch does not model the power consumption of the set selection multiplexers. As explained in Section 5, we do account for the energy consumption of additional Cool-Cache hardware blocks.

We consider two Cool-Cache configurations, SRAMs with 8-byte and 32-byte widths, and compare these against two traditional hardware caches, a direct mapped and a 4-way set associative cache. The results in Figure 14 are for 64K caches, the Cool-Cache has a hotline size of 256 bytes. As seen in the figure, Cool-Cache savings are higher for the 16-register configuration. The 32-byte-wide Cool-Cache achieves higher percentage energy savings than the 8-byte-wide Cool-Cache. Cool-Cache is substantially more energy efficient than not only the direct-mapped traditional cache but also the 4-way set-associative one. Note that the scratchpad-only energy savings are somewhat more sensitive to the register file size (Figure 6), whereas substantial energy savings are possible with the Cool-Cache even for an aggressively-sized register file, see Figure 14(b). This is due to the efficiency of the statically speculative hotlines component of Cool-Cache.

7. CONCLUSION

Our Cool-Cache framework achieves substantial energy savings for multimedia applications without compromising performance. Our research covers the architectural and compiler domains. We consider both scalars and non-scalars in our techniques and direct scalars into an energy-efficient minibuffer. We also propose and evaluate a new flexible compiler-controlled caching architecture that eliminates cache tags. The ideas presented in this paper could be applied for chipwide energy saving schemes as well. A natural extension of this work would be using statically speculative compiler-architectural methods to drive energy optimization at the fetch and issue stages.

REFERENCES

- ABRAHAM, S. G. AND MAHLKE, S. A. 1999. Automatic and efficient evaluation of memory hierarchies for embedded systems. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture, MICRO32*. Haifa, Israel.
- ALBONESI, D. H. 2000. Selective cache ways: On-demand cache resource allocation. *Journal of Instruction Level Parallelism*.
- BECHADE, R. ET AL. 1994. A 32b 66 Mhz 1.8 W microprocessor. In *Proceedings of the International Solid-State Circuits Conference*.
- BENINI, L., MACH, A., AND PONCINO, M. 2000. A recursive algorithm for low-power memory partitioning. In *Proceedings of the International Symposium on Low Power Electronics and Design, ISLPED00*, Rapallo, Italy.
- BROOKS, D., TIWARI, V., AND MARTONOSI, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture, ISCA'00*, Vancouver, Canada.

- BURGER, D. AND AUSTIN, T. D. 1997. *The simplescalar tool set, version 2.0*. Univ. of Wisconsin-Madison Computer-Sciences Department Tech. Rep. #1342.
- CALDER, B. AND GRUNWALD, D. 1994. Fast and accurate instruction fetch and branch prediction. In *Proceedings of the 21th International Symposium on Computer Architecture, ISCA'94*. Chicago, IL.
- CALDER, B. AND GRUNWALD, D. 1995. Next cache line and set prediction. In *Proceedings of the 22nd International Symposium on Computer Architecture, ISCA'95*. S. Marherita, Italy.
- CHIOU, D., JAIN, P., RUDOLPH, L., AND DEVADAS, S. 2000. Application-specific memory management for embedded systems using software-controlled caches. In *Proceedings of the 37th Design Automation Conference, DAC'00*. Los Angeles, CA.
- COOPER, K. D. AND HARVEY, T. J. 1998. Compiler-controlled memory. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Systems (ASPLOS-VIII)*.
- DELALUZ, V., KANDEMIR, M., VIJAYKRISHNAN, N., AND IRWIN, M. J. 2000. Energy-oriented compiler optimizations for partitioned memory architectures. In *Proceedings International Conference on Compilers, Architectures, and Synthesis for Embedded Systems CASES00*. San Jose, CA.
- HUANG, M., REANU, J., AND TORELLAS, J. 2001. L1 cache decomposition for energy efficient processors. In *International Symposium on Low-Power Electronics and Design, ISLPED'01*. Huntington Beach, CA.
- INTEL. 2000. *Intel StrongARM SA-1110 microprocessor brief datasheet*.
- KAMBLE, M. AND GHOSE, K. 1997. Analytical energy dissipation models for low power caches. In *International Symposium on Low Power Electronics and Design, ISLPED'97*. Monterey, CA.
- KIN, J., GUPTA, M., AND MANGIONE-SMITH, W. H. 1997. The filter cache: An energy efficient memory structure. In *Proceedings of the 30th Annual Symposium on Microarchitecture, MICRO30*.
- KULKARNI, C., CATTLOOR, F., AND MAN, H. D. 2000. Advanced data layout organization for multimedia applications. In *Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia (PDIVM 2000)*. Cancun, Mexico.
- LEE, C., POTKONIAK, M., AND MANGIONE-SMITH, W. H. 1997. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*.
- LEE, S. H. AND TYSON, G. S. 2000. Region-based caching: An energy efficient memory architecture for embedded processors. In *Proceedings of PACM (CASES'00)*. San Jose, CA.
- MEMIK, G., KANDEMIR, M., HALDAR, M., AND CHOUDHARY, A. 1999. A selective hardware/compiler approach for improving cache locality. Northwestern Univ. Tech. Rep. CPDC-TR-9909-016.
- MILUTINOVICH, V., TOMASEVIC, M., MARKOVIC, B., AND TREMBLAY, M. 1996. The split temporal/spatial cache: Initial performance analysis. In *Proceedings SCIZZL-5*. Santa Clara, CA.
- MONTENARO, J. ET AL. 1996. A 160 MHz 32b 0.5 W CMOS RISC microprocessor. In *Proceedings of the International Solid-State Circuits Conference*.
- MORITZ, C. A., FRANK, M., AND AMARASINGHE, S. 2001. Flexcache: A framework for compiler generated data caching. *Lecture Notes in Computer Science*. Springer-Verlag, Berlin.
- MORITZ, C. A., FRANK, M., LEE, W., AND AMARASINGHE, S. 1999. *Hot pages: Software caching for raw microprocessors*. MIT-LCS Tech. Memo LCS-TM-599.
- PANDA, P. R., DUTT, N. D., AND NICOLAU, A. 1997. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proceedings of European Design and Test Conference*. Paris, France.
- RANGANATHAN, P., ADVE, S., AND JOUPPI, N. P. 2000. Reconfigurable caches and their application to media processing. In *Proceedings of the 27th International Symposium on Computer Architecture ISCA'00*, Vancouver, Canada.
- UNSAI, O. S., ASHOK, R., KOREN, I., KRISHNA, C. M., AND MORITZ, C. A. 2001a. Cool-cache for hot multimedia. In *34th Annual International Symposium on Microarchitecture, MICRO34*.
- UNSAI, O. S., KOREN, I., AND KRISHNA, C. M. 2000a. Application level power-reduction heuristics in large scale real-time systems. In *Proceedings of the IEEE International Workshop on Embedded Fault-Tolerant Systems*. Washington, DC.
- UNSAI, O. S., KOREN, I., AND KRISHNA, C. M. 2000b. Power-aware replication of data structures in distributed embedded real-time systems. *Lecture Notes in Computer Science, LNCS2000*. Springer-Verlag, Berlin.

- UNSAI, O. S., KOREN, I., KRISHNA, C. M., AND MORITZ, C. A. 2002. The minimax cache: An energy-efficient framework for media processors. In *8th International Symposium on High-Performance Computer Architecture, HPCA8*. Cambridge, MA.
- UNSAI, O. S., WANG, Z., KOREN, I., KRISHNA, C. M., AND MORITZ, C. A. 2001b. On memory behavior of scalars in embedded multimedia systems. In *Workshop on Memory Performance Issues, 28th International Symposium on Computer Architecture, ISCA'01*. Goteborg, Sweden.
- WILTON, S. J. E. AND JOUPPI, N. P. 1996. Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits* 21, 5 (May).
- WITCHEL, E., LARSEN, S., ANANIAN, C. S., AND ASANOVIĆ, K. 2001. Direct addressed caches for reduced power consumption. In *Proceedings of the 34th Annual International Symposium on Microarchitecture, MICRO34*. Austin, TX.

Received February 2002; accepted July 2002