Cool-Cache for Hot Multimedia*

Osman S. Unsal, Raksit Ashok, Israel Koren, C. Mani Krishna, Csaba Andras Moritz

Department of Electrical and Computer Engineering, University of Massachusetts
E-mail: {ounsal,rashok,koren,krishna,moritz}@ecs.umass.edu

Abstract

We claim that the unique characteristics of multimedia applications dictate media-sensitive architectural and compiler approaches to reduce the power consumption of the data cache. Our motivation is exploring energy savings for real-time 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 caching architecture by re-evaluating 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.

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 [20] and Power PC [4], respectively. Therefore, if we save on the cache energy consumption, the overall energy consumption will also be considerably reduced.

In an earlier paper about the FlexCache project [23], we described our vision of a multipartitioned cache where memory accesses are 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. This is in line with our

philosophy that power and energy savings are inter-layer issues and that they should be addressed together all the way up from the architectural to the network layer [28, 29].
In particular, our contributions are:

- A compiler-controlled data remapping scheme directing scalar accesses to a small scratchpad SRAM area. Our scheme can be utilized in existing media processors and results in up to 38.2% average energy savings without sacrificing performance.

- *Hotlines*, a media-sensitive software-directed 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 no or minimal modifications to current architectures and compilers. In our previous work [27], we examined the memory footprint of scalars in multimedia applications and found them to be extremely small. However, we also established that a significant percentage of memory accesses in those applications are scalar accesses. This duality 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.

This paper is organized as follows. In Section 2 we analyze 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 energy. Providing architectural support to improve memory behavior include split caches which were discussed in [21]. Albonesi [2] proposed selective cache ways, a vertical cache partitioning scheme. Benini et al. [3] discuss an optimal SRAM partitioning scheme for an embedded system-on-a-chip. Panda et al. [24] propose use of a scratchpad memory in embedded processor applications. Kin et al. [13] study a small L0 cache that saves energy while reducing performance by 21%. Lee and Tyson [17] 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. [9] employ a software-controlled cache and use a cache way based partitioning scheme. A recent paper by Huang et al. [12] also uses 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 [19] have exclusively focused on arrays. A recent memory behavior study for multimedia applications has also primarily targeted array structures [15]. Another recent paper by Delazuz et al. [11] 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 [25], 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 [10] look at compiler-controlled memory. Their analysis includes spill memory requirements for some Spec '89 and Spec '95 applications. In this conference, Witchel et al. [31] propose a direct addressed cache which eliminates some cache tag accesses and thereby saves energy. In their study of instruction fetch prediction, Calder et al. [7] introduce a tagless memory buffer for next cache line and set prediction. Abraham and Mahlke [1] evaluate memory hierarchies for embedded systems from a performance point of view.

Our previous work [22, 23, 27, 28, 29] 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 can be used in existing architectures: remap every scalar memory access into a scratchpad memory area. 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 an SRAM scratchpad area in the Samsung ARM7 and Hitachi SH2. The recently introduced Intel StrongARM SA-1110 [14] has a 512 byte minicache for frequently used data. In our previous study of the Mediabench benchmarks [27], 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 non-partitioned architecture. In fact, since we decrease the cache interference, we get better data cache performance by separating scalar accesses from array accesses [27]. 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 which 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 et al. [8] 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.

A 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 disambiguation between the scalar and non-scalar accesses. As described in [27], although the scalars typically have a very small footprint, they are frequently accessed, and have considerable interference with non-scalar accesses. The Cool-Cache architecture, by statically diverting the scalar and non-scalar 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 ac-

cessed 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 non-scalar 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 additional runtime overhead is required. The non-scalars however, if managed explicitly in the compiler, 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 for during fills associated with cache misses. This translation can be done by inserting a sequence of compiler generated instructions, at the expense of added software overhead. But as discussed in [22], for many applications, there is a lot of 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 [23], 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 miss-predicted access) we use a small 16-entry fully associative *cache TLB* to cache address mappings for memory accesses that are miss-predicted. We found that a 16-entry *cache TLB* 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, to minimize address translation overhead in virtual memory systems.

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. 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 1 gives 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 1Kbyte structure in our experiments) compared to the regular SRAM data array.
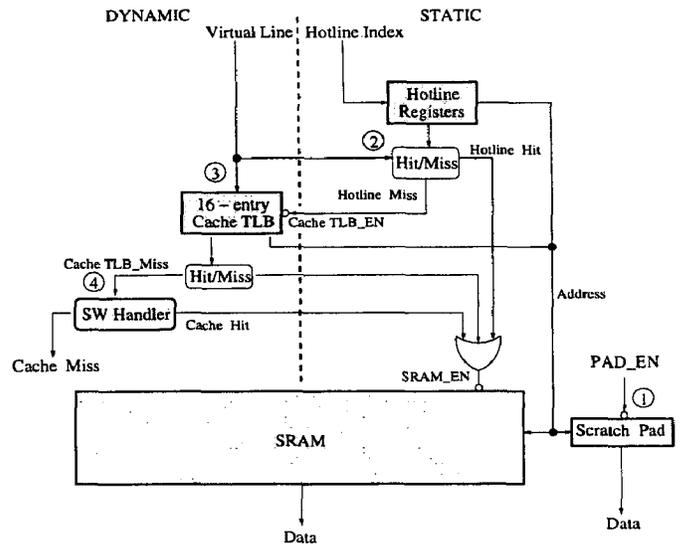

Figure 1. The Cool-Cache Architecture

The non-scalar 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 itself is stored in a non-mapped portion of the SRAM) to check if it is a cache hit/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, the inter-

ference between scalar and non-scalar accesses has been eliminated resulting in higher hit-rate, and better cache utilization. Secondly, 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 [30], the tag-access is on the critical path and can add as much as 30% to access time of associative caches. So, 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 [30]. 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 non-scalar accesses. Second, the SRAM is used as a simple addressable memory - the complicated access mechanisms of a regular cache consume more power.

As shown by the results, 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 a recent paper by Delaluz et al. [11] for a discussion of energy conscious interleaved memories.

4. Cool-Cache Compiler

The overall complexity of the Cool-Cache compiler is not much higher than that of a regular compiler. Figure 2 shows a high-level picture of the stages involved. The
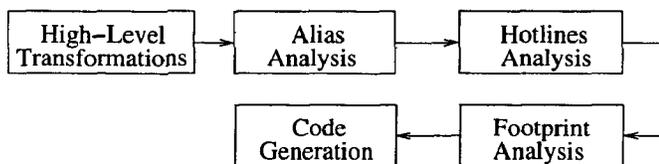

Figure 2. Cool-Cache compiler stages

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 isn't so 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 the FlexCache paper, because alias analysis information is disregarded. Algorithm 1 shows the pseudocode. The scalar footprint analysis [27] 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: they 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[16] in our experiments. See Table 1 for a short description of the benchmarks included in our analysis.

Figure 3 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 modifications. With this in mind, we chose the SUIF/Machsuif suite as our compiler framework. SUIF [26] does high-level passes while Machsuif [18] makes machine specific optimizations. We have modified SUIF/Machsuif passes for our memory remapping schemes and 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 non-scalar 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 [5] tool suite to run the bi-

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	OpenGL 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

Table 1. Applicable Mediabench benchmarks. We do not include PGP, a public key encryption scheme which is very similar to PEGWIT. We also do not include GHOSTSCRIPT since it is more amenable to embedded systems than multimedia.
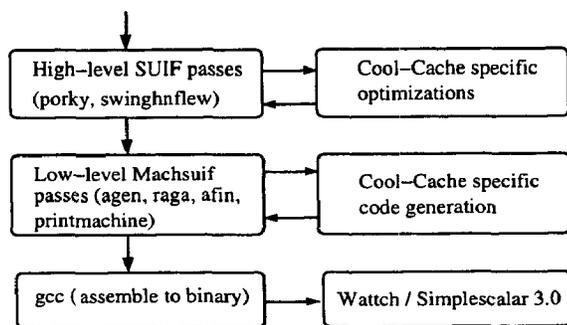
Figure 3. Experimental Setup Block Diagram

naries and collect the energy results. Wattch is based on the Simplescalar [6] framework. The simulators have been modified to recognize the annotations in the *marked* code, do hotline register checks, *cache TLB* checks, etc.. 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, etc. are output by the simulators.

Our baseline machine model is an ARMlike single-issue in-order processor. Lee et al. [17] 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 2. We use the activity sensitive conditional clocking power model in Wattch, i.e., the cache consumes power 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 which includes leakage power: the absence of data cache tags in our framework which

Hardware Block Name	Modeled as	Power (W.)
64bit Wide L1 Memory	SRAM	6.86
256bit Wide L1 Memory	SRAM	5.58
1Kb Scratchpad Memory	SRAM	0.68
8 Hotline Registers	Register File	0.16
16 Entry Cache TLB	CAM	0.34

Table 2. Cool-Cache specific hardware power consumption. Note that L1 data memory is tagless.

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 3. Data cache sizes for typical media processors.

would otherwise consume static power lead to higher savings. To determine the baseline cache size, we did a survey of data cache sizes of current multimedia processors. As Table 3 indicates, the trend is towards larger caches. Therefore we have selected a 64Kbyte 2-way cache as our baseline. We also examine 32K and 128K caches in our sensitivity analysis. See Table 4 for the baseline configuration.

5.1. Scalar Data Remapping

Our main focus is Machsuif's register allocator pass, Raga. Raga determines the register allocation. The allo-

278

Algorithm 1 The Hotline Algorithm

```
/* For each routine, start the annotation process by start-
ing 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
```

cation 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 are smaller than the scratchpad area. We presented the compiler algorithm that extracts the footprint size in [27].

6. Results

6.1. Scratchpad Energy Savings

Unless otherwise stated, all the results in this section are with a scratchpad of size 1024 bytes, and the baseline cache is 64Kbyte 2-way associative. We ran the benchmarks using the modified Wattch/Simplescalar and collected the data cache energy results. Figure 4 shows the percentage energy savings for our 32 general purpose register media proces-

Processor Speed	1GHz
Process Parameters	0.35 micron, 2.5V
Issue	In-order Single-issue
L1 D-cache	64Kb, 2-way associative
L1 I-cache	32Kb, 2-way associative
Scratchpad	1Kb
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

Table 4. Baseline Parameters.

sor model. 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 shown in Figure 4. Our technique saves an average of 38.2% of energy.

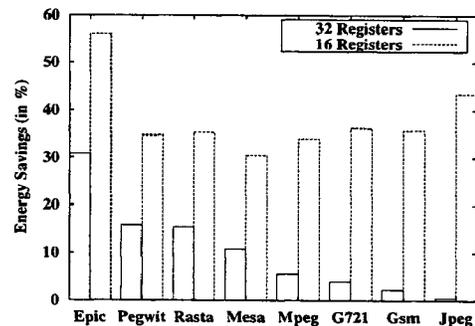In fact, we show that we can be just as energy-efficient

Figure 4. Scratchpad Energy Savings.

with a 16-register media processor with a scratchpad SRAM as a 32-register processor with no scratchpad, see Figure 5. Actually, the overall energy savings are even greater 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 a larger one that exercises the caches more. We used this alternative input set and ran the applications for a study of input sensitivity. Although the data cache energy consumption of the second set is higher, the results in Figure 6(a) suggest that the energy savings are independent of the input sets included in the Mediabench.

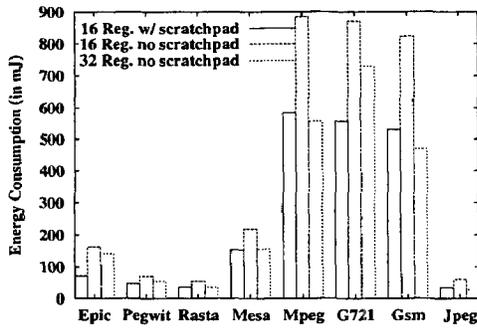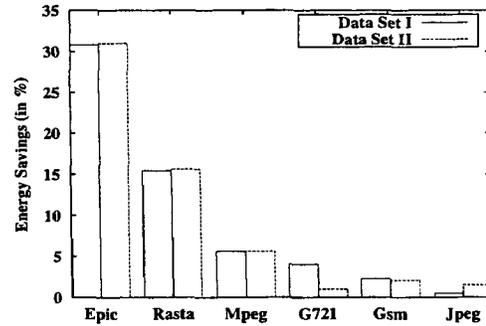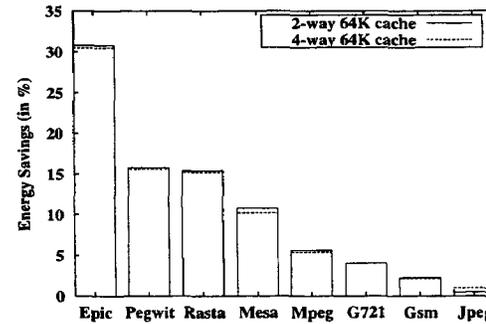Next we explore the sensitivity of the energy savings

279

Figure 5. 16-register architecture with scratchpad can be more energy efficient than 32-register architecture without scratchpad.

(a) Sensitivity to input

(b) Sensitivity to cache associativity

Figure 6. Energy savings sensitivity analysis

on cache associativity and size. We compared our baseline cache with a 64K 4-way cache. The results in Figure 6(b) show that the savings are fairly independent of cache associativity. We have also looked at the impact of cache size. Figure 7(a) shows the energy consumption in millijoules for three cache sizes. Figure 7(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.

6.2. Cool-Cache Performance and Energy Savings

6.2.1 Prediction rates

The prediction rates of the hotlines scheme are shown in Figure 8. The sensitivity of the prediction rates to both cache line size and cache size are also shown. Figure 8(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 32Kb, 64Kb, and 128Kb, respectively. Figure 8(b) has the cache size fixed at 32Kb with line sizes of 1024b, 256b, and 64b.

From the second graph, 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, because the media applications exhibit high spatial locality, even a line-size as large as 1Kb does not degrade cache performance (except for pegwit, where the cache-miss rate is seen to drop as line size decreases). 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.

The first graph shows that the static and dynamic prediction rates are almost independent of cache size. The prediction rate is dependent 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 256byte wide line, this translates to 6kbytes. Therefore, for cache sizes above, say 8Kb, the prediction rate will be fairly constant. Figure 9 shows the sensitivity to the input data size, where the cache and line sizes have been fixed at 32kb and 1024bytes, 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.

6.2.2 Performance

We now study the impact of the Cool-Cache on performance. Figure 10 shows the memory performance, i.e, the cycles spent on memory instructions (values are normalized to a scale of 0-1). There are four stacked bars for each application. The leftmost bar corresponds to a hardware cache - the bottom bar is the time spent on hits, the upper bar is that spent on misses. The next three bars are for Cool-Caches
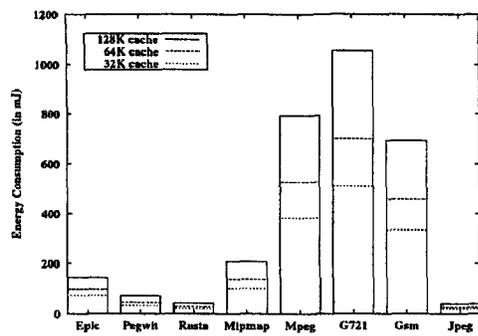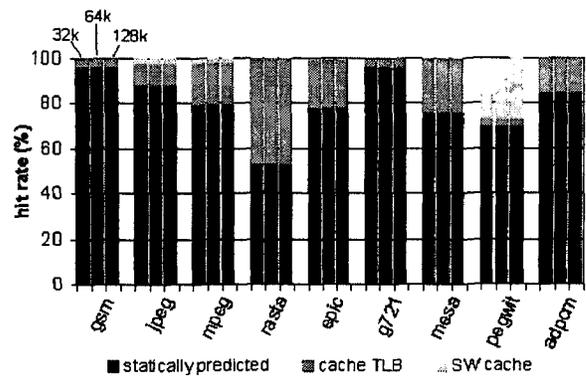
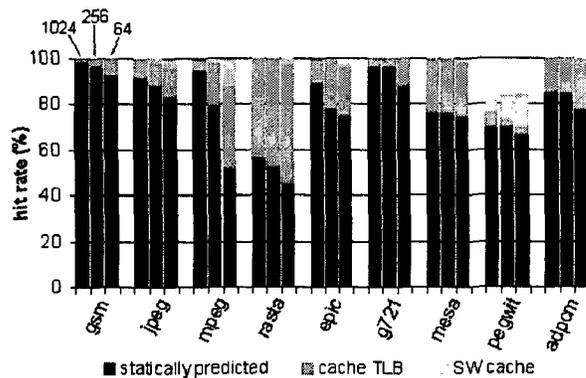(a) Energy consumption
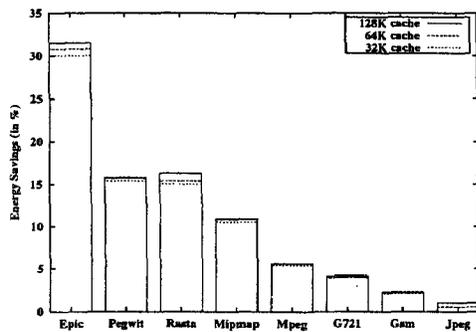
(b) Energy savings

Figure 7. Energy consumption and savings for different cache sizes

(a) 256 byte line

(b) 32K Cache

Figure 8. Hit Rate for different configurations

of size 32kb and line sizes of 1024b, 256b, and 64b, respectively. Each bar has several components. Starting from the bottom, they are: time spent on scratchpad accesses, on 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 are based on cycle counts and not time - as already mentioned, the simpler access mechanism in Cool-Cache as compared to a hardware cache, can lead to shorter cycle times.

Figure 11 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 bench-

marks 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 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 higher: 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 also account for the energy consumption of additional Cool-Cache hardware blocks as well.

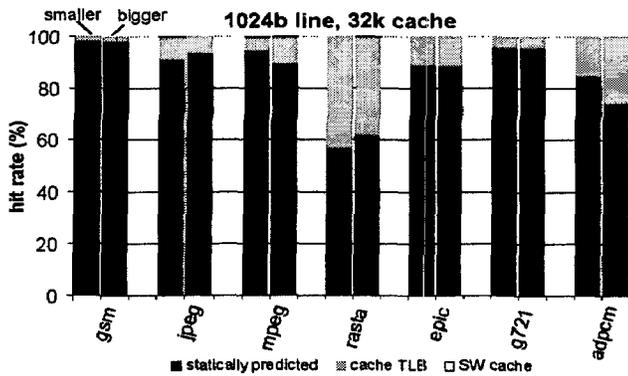We consider two Cool-Cache configurations, SRAMs with
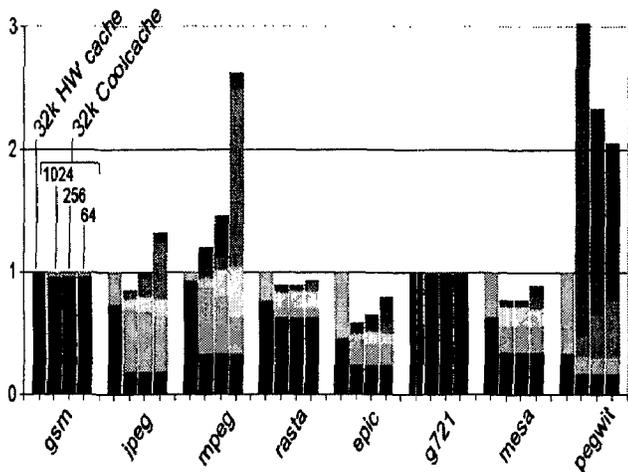
281

Figure 9. Hit rate sensitivity to benchmark input

Figure 10. Cool-Cache memory performance

Figure 11. Cool-Cache performance

and evaluate a new flexible compiler-controlled caching architecture that eliminates cache tags.

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 12 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 width Cool-Cache achieves higher percentage energy savings than the 8-byte width Cool-Cache. Note that Cool-Cache is substantially more energy efficient than not only the direct mapped traditional cache but also the 4-way set associative one.

7. Conclusion

We present our Cool-Cache framework that acieves 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. In particular, we propose
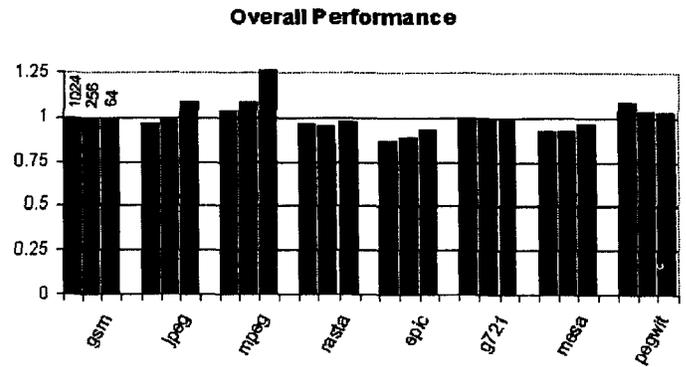
References

[1] Abraham S. G., Mahlke S. A., "Automatic and Efficient Evaluation of Memory Hierarchies for Embedded Systems," *Proceedings of the 32nd Annual International Symposium on Microarchitecture, MICRO32*, Haifa, Israel, November 1999

[2] Albonesi D. H., "Selective Cache Ways: On-Demand Cache Resource Allocation," *Journal of Instruction Level Parallelism*, May 2000

[3] Benini L., Macii A., Poncino M., "A Recursive Algorithm for Low-Power Memory Partitioning," *Proceedings of the International Symposium on Low Power Electronics and Design, ISLPED00*, Rapallo, Italy, 2000

[4] Bechade R. et al., "A 32b 66MHz 1.8W Microprocessor," *Proceedings of the International Solid-State Circuits Conference*, 1994

[5] Brooks D., Tiwari V., Martonosi M., "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proceedings of the 27th International Symposium on Computer Architecture, ISCA'00*, Vancouver, Canada, June 2000

[6] Burger D., Austin T. D., "The Simplescalar Tool Set, Version 2.0," *University of Wisconsin-Madison Computer-Sciences Department Technical Report #1342*, June 1997

[7] Calder B., Grunwald D., "Next Cache Line and Set Prediction", *Proceedings of the 22nd International Symposium on Computer Architecture, ISCA'95*, S. Marherita, Italy

[8] Calder B., Grunwald D., "Fast and Accurate Instruction Fetch and Branch Prediction", *Proceedings of the 21th International Symposium on Computer Architecture, ISCA'94*, Chicago, IL, April 1994

[9] Chiou D., Jain P., Rudolph L., Devadas S., "Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches," *Proceedings of the 37th Design Automation Conference, DAC'00*, Los Angeles, CA, June 2000

[10] Cooper K. D., Harvey T. J., "Compiler-Controlled Memory," *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Systems (ASPLOS)* October, 1998

282
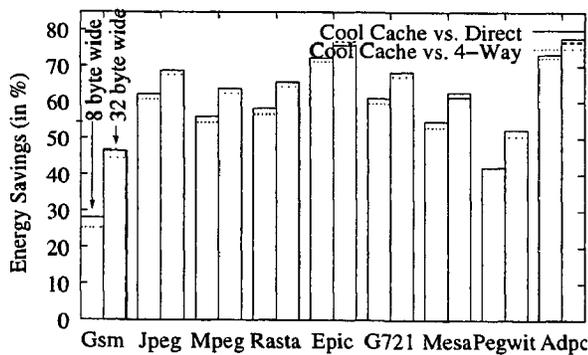
(a) 8-Byte and 32-Byte wide SRAM Cool-Cache 16-register CPU
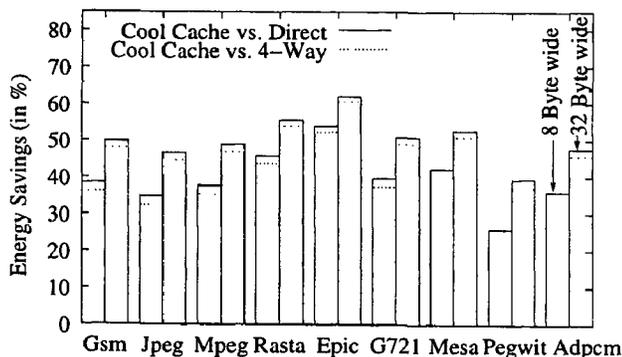
(b) 8-Byte and 32-Byte wide SRAM Cool-Cache 32-register CPU

Figure 12. Cool-Cache energy savings

[11] Delaluz V., Kandemir M., Vijaykrishnan N., Irwin M. J.,"Energy-Oriented Compiler Optimizations for Partitioned Memory Architectures," *Proceedings International Conference on Compilers, Architectures, and Synthesis for Embedded Systems CASES00*, San Jose, CA, November 2000

[12] Huang M., Reanu J., Torellas J., "L1 Cache Decomposition for Energy Efficient Processors," *International Symposium on Low-Power Electronics and Design, ISLPED'01*, Huntington Beach, CA, August 2001

[13] Kin J., Gupta M., Mangione-Smith W. H., "The Filter Cache: An Energy Efficient Memory Structure," *Proceedings of the 30th Annual Symposium on Microarchitecture, MICRO30*, 1997

[14] *Intel StrongARM SA-1110 Microprocessor Brief Datasheet*, April 2000

[15] Kulkarni C., Catthoor F., H. De Man, "Advanced Data Layout Organization for Multi-media Applications," *Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia (PDIVM 2000)*, Cancun, Mexico, May 2000

[16] Lee C., Potkonjak M., Mangione-Smith W. H., "Mediabench: A Tool for Evaluating and Synthesizing Multime-

dia and Communications Systems," *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997

[17] Lee S. H., Tyson G. S., "Region-Based Caching: An Energy Efficient Memory Architecture for Embedded Processors," *Proceedings of PACM (CASES'00)*, San Jose, CA, November 2000

[18] http://www.eecs.harvard.edu/hube/software/software.html

[19] Memik G., Kandemir M., Haldar M., Choudhary A., "A Selective Hardware/Compiler Approach for Improving Cache Locality," *Northwestern University Technical Report CPDC-TR-9909-016*, 1999

[20] Montenaro J. et al., "A 160MHz 32b 0.5W CMOS RISC Microprocessor," *Proceedings of the International Solid-State Circuits Conference*, 1996

[21] Milutinovich V., Tomasevic M., Markovic B., Tremblay M., " The Split Temporal / Spatial Cache: Initial Performance Analysis," *Proceedings SCIzzL-5*, Santa Clara, California, March 1996

[22] Moritz C. A., Frank M., Lee W., Amarasinghe S., "Hot Pages: Software Caching for Raw Microprocessors," *MIT-LCS Technical Memo LCS-TM-599*, Aug 1999

[23] Moritz C. A., Frank M., Amarasinghe S.,"FlexCache: A Framework for Compiler Generated Data Caching," *To appear in Lecture Notes in Computer Science, Springer-Verlag*, 2001

[24] Panda P. R., Dutt N. D., Nicolau A., "Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications," *Proceedings of European Design and Test Conference*, Paris, France, 1997

[25] Ranganathan P., Adve S., Jouppi N. P., "Reconfigurable Caches and Their Application to Media Processing," *Proceedings of the 27th International Symposium on Computer Architecture ISCA'00*, Vancouver, Canada, June 2000

[26] http://suif.stanford.edu/

[27] Unsal O. S., Wang Z., Koren I., Krishna C. M., Moritz C. A.,"On Memory Behavior of Scalars in Embedded Multimedia Systems," *Workshop on Memory Performance Issues, 28th International Symposium on Computer Architecture, ISCA'01*, Goteborg, Sweden, June 2001

[28] Unsal O. S., Koren I., Krishna C. M., "Power-Aware Replication of Data Structures in Distributed Embedded Real-Time Systems," *Lecture Notes in Computer Science, LNCS2000 Springer-Verlag*, May 2000

[29] Unsal O. S., Koren I., Krishna C. M., "Application Level Power-Reduction Heuristics in Large Scale Real-Time Systems," *Proceedings of the IEEE International Workshop On Embedded Fault-Tolerant Systems*, Washington, DC, September 2000

[30] Wilton S. J. E., Jouppi N. P., "CACTI: An enhanced Cache Acess and Cycle Time Model," *IEEE Journal of Solid-State Circuits*, vol.21, no.5, May 1996

[31] Witchel E., Larsen S. Ananian C. S., Asanović K., "Direct Addressed Caches for Reduced Power Consumption", *Proceedings of the 34th Annual International Symposium on Microarchitecture, MICRO34*, Austin, TX, December 2001