The Minimax Cache: An Energy-Efficient Framework for Media Processors*

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

Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003
E-mail: {*ounsal,koren,krishna,moritz*}*@ecs.umass.edu*

Abstract

This work is based on our philosophy of providing inter-layer system-level power awareness in computing systems [26, 27]. Here, we couple this approach with our vision of multipartitioned memory systems [18, 19, 25], where memory accesses are separated based on their static predictability and memory footprint and managed with various compiler controlled techniques.

We show that media applications are mapped more efficiently when scalar memory accesses are redirected to a minicache. Our results indicate that a partitioned 8K cache with the scalars being mapped to a 512 byte minicache can be more efficient than a 16K monolithic cache from both performance and energy point of view for most applications. In extensive experiments, we report 30% to 60% energy-delay product savings over a range of system configurations and different cache sizes.

1. Introduction

The caching subsystem in the recently introduced low-power media/embedded processors consume a significant portion of the total processor power: 42% and 23% in StrongARM 110[17] and Power PC[2], respectively. Therefore, if we save on the data cache energy consumption, the overall energy consumption will also be considerably reduced. We firmly believe that the unique characteristics of multimedia applications dictate media-sensitive inter-layer architectural and compiler approaches to reduce the power consumption of the data cache. Our previous work extracted these characteristics; here we leverage this information to form our energy-saving Minimax cache framework.

Simply put, the Minimax cache is a regular L1 data cache with an additional small memory area. This small memory area is implemented as a statically managed *minicache*. While the size of the minicache is kept to a *minimum*, we employ it to get *maximum* energy-performance benefits. This organization leverages the fact that many accesses have very small footprints but they are frequently accessed. In our previous work, we have determined that scalar accesses exhibit this behavior for media applications[25]. We therefore redirect the scalar memory accesses to the minicache. Statically diverting the scalar and non-scalar accesses to the minicache and the regular L1 data cache, respectively, not only eliminates the cache interference but also saves power by only accessing a small minicache instead of a much larger data-array. In our previous work [25], we examined the interference issues. Here, we concentrate on the power aspect. Our results for a range of media applications indicate a 30% to 60% improvement in the energy-delay product.

The research spans the compiler and architectural layers. Our compiler-level analysis separates scalars from non-scalars and we map them to separate cache partitions. Existing cache partitioning schemes are driven by architectural features; examples are partitioning along instructions/data or along stack/heap accesses. The unique aspect of our Minimax cache partitioning scheme is the fact that it is driven by an application feature: scalar accesses. This is not only a finer granularity scheme but also one that blurs the compiler/architecture interface. By taking cache partitioning closer to the application layer, we are shifting caching from a hardware concern to a hardware/compiler one.
Briefly, our contributions in this paper are:

- Introduction of a new compiler-level scheme to partition scalars into a separate, much smaller cache area: the *minicache*. The scalars in the minicache and the non-scalars in the regular cache form the Minimax cache framework.
- Presentation of extensive experimental results on media applications and demonstrating that the Minimax cache organization is substantially more energy efficient while exhibiting high performance.
- Analyzing the energy impact of varying the register file size on media applications. We use register file sizes of 16 and 32.

*This research was supported in part by the National Science Foundation under grant EIA-0102696

Note that the Minimax cache approach is different from techniques to reduce energy dissipation that are based on placing a small cache in *front* of the L1 cache and managing it dynamically. Those approaches can come with a significant performance degradation relative to a conventional cache due to an increase in the access time on a miss in the small cache. Kin et al.[10] study a small L0 cache, called the *filter* cache, that saves energy while reducing performance by 29%.

The rest of this paper is organized as follows: We provide an analysis of related work in Section 2. Section 3 discusses our approach in more detail and provides our motivation. In Section 4 we introduce our experimental methodology. The results are given in Section 5. We conclude with a brief summary and a synopsis of future work in Section 6.

2. Previous Work

Previous cache partitioning research focused more on performance issues rather than energy. Providing architectural support to improve memory behavior include vertical cache partitioning schemes such as the selective cache ways proposed by Albonesi[1]. Panda et al.[20] propose use of a scratchpad memory in embedded processor applications. Kin et al.[10] study a small L0 cache that saves energy while reducing performance by 21%. Lee and Tyson [14] use the mediabench benchmarks and have a coarse-granularity partitioning scheme: they opt for dividing the cache along OS regions for energy reduction. A recent paper by Huang et al. [8] has a separate partition for the stack, they also address compiler implementation concerns as well.

Combined compiler/architectural efforts toward increasing cache locality [16] have exclusively focused on arrays. A recent memory behavior study for multimedia applications has also primarily targeted array structures [12]. Another recent paper by Delazuz et al. [7] 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, has considered dynamically dividing caches into multiple partitions [22], using the Mediabench benchmark in the performance analysis, with comments on compiler controlled memory. Cooper and Harvey [6] look at compiler-controlled memory. Their analysis includes spill memory requirements for some Spec '89 and Spec '95 applications. Sanchez et al. [23] introduce a compiler interference analysis and use a dual data cache for programs that have a high conflict miss ratio.

The fusion of the above research provided the motivation for this work. We consider scalar memory accesses, not only array or spill memory accesses, and we target multimedia systems running a suite of media applications. Compared to other cache partitioning approaches, we adopt a compiler-managed, application-driven finer granularity scheme: we direct scalar memory accesses to a minicache. To be fair, we do an execution time analysis of our minicache-added caching framework with a larger cache.

3. Background and Motivation

Our focus in this paper is multimedia architectures, but the methodology described can be applied to other classes of applications. We use the recently developed Mediabench benchmarks [13] in our experiments. Mediabench is a collection of popular embedded applications for communications and multimedia. See Table 1 for a short description of the benchmarks included in our analysis.

In our previous work [25], we implemented a compiler/architectural level scalar/non-scalar partitioning scheme through the use of a minibuffer. We define scalar accesses to be original and compiler-introduced scalar variables that couldn't be register promoted. One conclusion that was drawn in that study was that the scalars in media applications have a very low memory footprint and high access frequency. They also have considerable interference with non-scalar accesses. For the sake of clarity, we replicate the results for scalar static memory footprint upper bound using a 32 register configuration, see Table 2. Depending on the application, the actual memory footprint can be much smaller: addresses associated with variables that are no longer live could be reused, dead code sections could be eliminated, and so on. A compiler algorithm to compute the actual footprint requires inter-procedural analysis and a complex data-flow extraction. We have devised an effective compiler-level approximation for this purpose, the details and results are in [25].

The Minimax cache framework uses the same partitioning scheme, however we map into a minicache instead of a

Application	(In Bytes)
ADPCM	0
EPIC	321
G721	48
GSM	202
JPEG	502
MESA	2191
MPEG	2125
RASTA	618

Table 2. The Memory Size Requirements.

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
RASTA	Speech recognition front-end processing

Table 1. Applicable Mediabench benchmarks. We do not include GHOSTSCRIPT since it is more amenable to embedded systems than multimedia. We also do not include the public key encryption schemes, PGP and PEGWIT, for similar reasons.

minibuffer and we conduct an extensive energy analysis.

3.1. Compiler and Architectural Implementation

Many embedded and media processors already have a scratchpad minibuffer or a minicache. The difference between the two organizations is that the minibuffer approach would require explicit management at compile-time of the memory accesses that are mapped into it, a conservative approach because compile-time analysis typically would need to overestimate the footprint to match exactly the size of the minibuffer, while the minicache could be a fully associative cache where the compile-time estimated footprint could be larger than the cache size. In practice, the dynamic memory footprint of memory accesses mapped into the small area is likely to be smaller than the static estimate, resulting in a high hit-rate in the minicache even if its size is chosen somewhat smaller than the static footprint size. Additionally, this latter approach could be used together with a simpler compiler analysis that gives a larger lower bound on the dynamic footprint. The minibuffer approach would require an exact bound on the footprint size to guarantee program execution correctness but, if that is successfully done would consume somewhat less power than the minicache. Coherence between the L1 data and minicache is guaranteed through type information based partitioning. In case one would want to add other data accesses to the minicache then alias analysis should be used. The output of the alias analysis would determine if there is any overlap in between the different memory accesses. If there is overlapping, mapping to the minicache should be avoided.

As far as implementation is concerned, no architectural modifications are necessary if the media processor is equipped with a minicache. An example is the recently introduced Intel StrongARM SA-1110 [9] which has a 512 byte minicache, see Figure 1. This processor is very similar to our baseline. If the media processor is not equipped with any kind of minicache mechanism, then assembler annotations can be used to devise special load/store instructions which would channel the scalar data to a separate, smaller cache area.

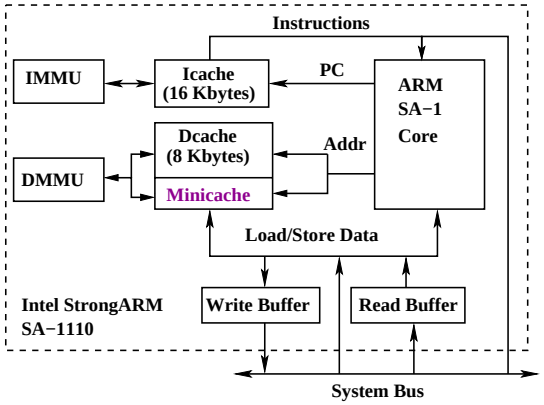


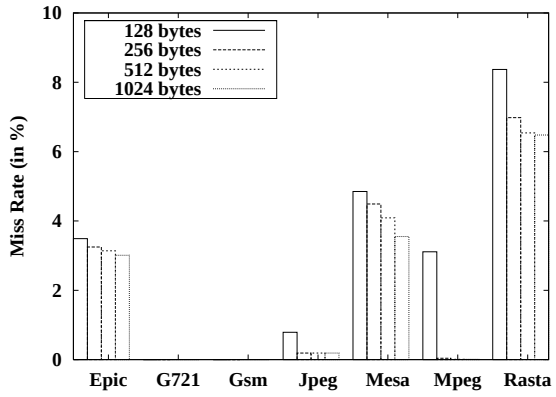
Figure 1. Intel Strongarm SA-1110 Architecture.

3.2. Feasibility Study

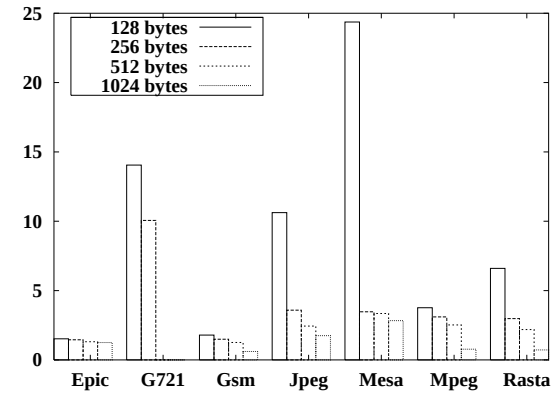
The minibuffer approach[25] requires sophisticated compiler-level analysis. Our new framework, the Minimax cache, avoids most of this complexity. Since the footprint of scalars is low, we expect the Minimax cache to be as efficient as the minibuffer approach. One way of ascertaining

this is to analyze how well the scalars are mapped into the separate cache area. In Figure 2, we present the minicache miss rates. The associated experimental setup is described in Section 4. We provide the results for both a 16 register and a 32 register processor, the smaller register file increases the number of scalar memory accesses. Note that a very small minicache size of 128 bytes can hold the working set of the scalar data for most applications. The working set of the 16 register configuration is higher due to the additional spills. However, even for a 16 register combination a size of 512 bytes is sufficient.

These results verify the promise of our approach. Although the Minimax cache requires some additional chip area compared to the monolithic cache, the impact is negligible since the minicache sizes used in this work are very small. Moreover, our comparative performance study of the Minimax Cache with a much larger monolithic cache indicate that Minimax cache can even outperform the larger cache: see Section 5.



(a) 32-register processor



(b) 16-register processor

Figure 2. Miss Rates for different minicache sizes.

4. Experimental Methodology

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 [24] does high-level passes while Machsuif [15] makes machine specific optimizations. First, all the source files are converted into SUIF format and merged into one SUIF file. Next, we run this SUIF file through the Machsuif passes. We modified Machsuif passes to annotate all the scalar accesses and to vary the machine register file size. The modifications are propagated using the SUIF annotation mechanism. We amend the resulting assembler code by inserting NOP-like instructions around the annotated scalars, thus *marking* them.

We then used the Wattch [4] tool suite to run the binaries and collect the energy results. Wattch is based on the Simplescalar [5] framework. The simulators has been modified to recognize the annotations in the *marked* code.

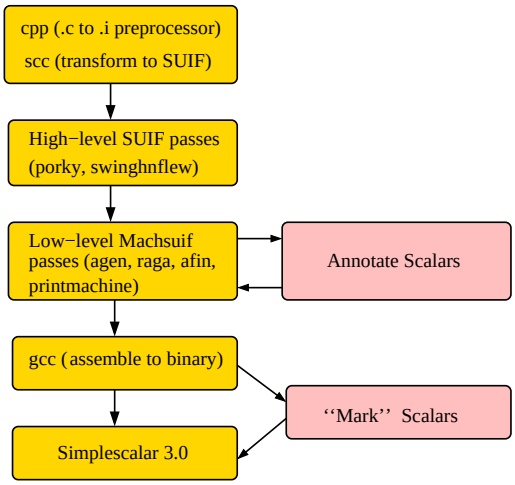


Figure 3. The Experimental Setup Block Diagram

Our baseline machine model is an ARM-like single-issue in-order processor. Lee et. al. [14] use an identical configuration in their power dissipation analysis of region-based caches for embedded processors. We use the activity sensitive conditional clocking power model in Wattch, i.e., the cache consumes power when it is accessed. We modified Wattch to calculate the energy consumption of the addi-

Processor	L1 Cache	L2 Cache
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
Intel StrongArm 110	16K	None
Intel StrongARM 1100	8K	None

Table 3. Cache configurations for typical media processors.

tional minicache. To determine the baseline architecture, we did a survey of current media processors. As Table 3 indicates, the trend is towards larger caches. Therefore we have selected a 64Kbyte 2-way cache as our baseline, see Table 4. To be fair, we also examine 8K caches as well. The table also indicates that media processors do not typically have L2 data caches. Therefore, we only have L1 caches in our baseline architecture. However, some recent media processors have L2 data caches, the Transmeta Crusoe TM5400 is one example. So we also include an analysis of energy-delay impact of L2 caches. Here a concern might be the issue of consistency between the L2 cache and the L1 data + minicache. Namely, the block fetched from L2 into the L1 caches could contain a mix of scalar/non-scalar data. We avoid this problem by keeping the block sizes same across the caches. If the block sizes were different, then the issue could be addressed by clustering the scalar data to the beginning of the address space and padding them appropriately to the size of the L2 cache block size and boundary.

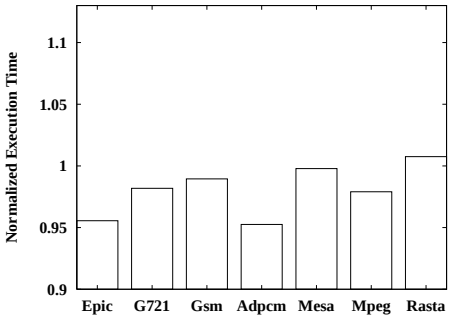
Processor Speed	1GHz
Issue	In-order Single-issue
L1 D-cache	64Kb, 2-way associative
Minicache	256bytes, fully-associative
L1 I-cache	32Kb, 2-way associative
L2 cache	None
L1 D-cache hit time	2 cycles
Minicache hit time	1 cycle
L2 cache hit time	20 cycles
Main memory hit time	100 cycles

Table 4. Baseline Parameters.

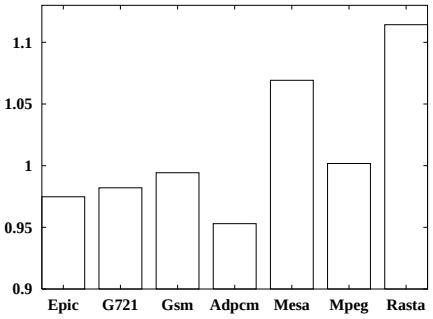
5. Results

Separating frequently-accessed low-footprint scalars from non-scalars by mapping them into separate cache areas decreases cache interference. This leads to a more efficient use of the available caches. To underscore that intuition and to be fair, we next show that a much smaller 8Kbyte+512 byte Minimax cache can outperform a 16Kbyte monolithic cache for media applications, only for a few applications the larger cache is better, see Figure 4. The analysis is done for configurations with and without an L2 cache. We only include the execution time results here since the energy efficiency of the smaller Minimax cache is trivially obvious. The implication is that compiler-level schemes can result in saving chip real-estate by partitioning application accesses.

When introducing energy saving methods we need to be conscious of performance impacts of the proposed



(a) With 256K L2 Cache



(b) No L2 Cache

Figure 4. Depending on benchmark, a Minimax Cache of much smaller size can be better than a monolithic cache of larger size. Here the results are the normalized execution time of an 8Kbyte Minimax cache with a 512 byte minicache. They are normalized with respect to the execution time of a 16Kbyte monolithic cache. To be fair, configurations with and without an L2 level cache are shown.

method: we might save energy but this can come at the expense of performance. We therefore use the energy-delay product as our metric in our experiments. Here the delay is expressed as the total execution time. Figure 5 after the References shows the energy-delay product results in terms of $microJoules \cdot second$ for the L1 caching subsystem for the mediabench applications. We take the most challenging configuration for the Minimax cache: 16 register processor without L2 cache to penalize Minimax cache capacity misses. The results compare 8K and 64K monolithic caches with the same size Minimax cache with a minicache size of 128, 256, 512 and 1024 bytes. We get significant savings for the 64K cache: close to 30% for the worst case. For some applications with a very small footprint but frequent scalar accesses the savings are even more pronounced, 60% for the Epic benchmark for a minicache size of 128 bytes. The sensitivity of the results to the minicache size exhibits some interesting behavior. For some benchmarks the energy-delay product decreases as we increase the minicache size and then starts increasing as the size is further increased. This is due to the complex interplay between energy and performance: for those benchmarks small minicache sizes incur more scalar misses affecting performance, as the minicache size is increased the working set of the scalars fit in the cache and there is an *optimum* point beyond which the energy-delay product starts to increase with the increased energy consumption dominating the product. For some applications such as the Mesa or Rasta, a 128 byte minicache size has a higher energy-delay product than the monolithic cache since the working set of the scalars does not fit in 128 bytes. However, even for those applications a minicache of either 256 or 512 bytes gives substantial savings.

As the main memory to processor speed discrepancy grows, L3 caches have become feasible for general purpose processors[21]. Following the same trend, some recent media processor designs have started to include on-chip L2 caches, see Table 3. The rationale for this is the widening memory-gap as much as multiprogramming related causes. This motivates us to look into the combined energy-delay signature of the L1 data cache together with the L2 cache. We use a 256Kbyte 4-way associative L2 cache for this study. Figure 6 shows the results. Note that the energy-delay issues related with scalar data not fitting in the minicache for such benchmarks as Mesa or Rasta has been masked by the L2 cache. Comparison of Figure 6 with Figure 5 provides other interesting insights: note that the L1+L2 combination is more energy-delay effective for most benchmarks than L1 only configuration, even with the additional energy consumption of the L2 cache taken into account. This observation implies that future media processors can benefit from an on-chip L2 cache.

There is another aspect of scalar access related energy consumption: the register file. Since we model a media processor that is akin to the ARM architecture, we have used a register file size of 16 in our experiments. We now extend our analysis to register related energy impact of scalars and vary the register file size. Compared to the 32-register processor, average power per register access is lower for the 16-register case. This might imply that register file energy consumption of the 16-register processor would be less than 32-register processor for most applications. However, Figure 7 shows a counter-intuitive result: a 16-register processor consumes more register related energy than a 32-register processor for the Adpcm application. This is because all scalars for the Adpcm application fit in 32 registers and there are no scalar related memory accesses, see Table 2. However, the 16 register file is too small for the scalars to fit in and some scalars spill to memory. The energy impact of those additional scalar related data transfers between the registers and memory cause the 16-register processor to consume more register-related energy than 32-register one. This phenomenon offers the following insight: compiler/architecture coupling is becoming stronger and should be considered at the design stage.

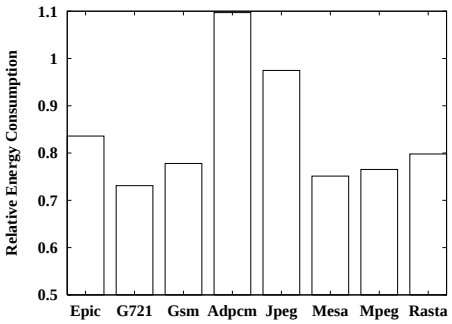
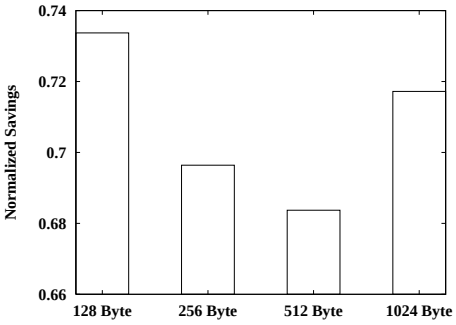


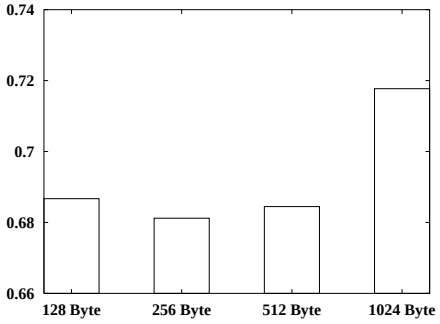
Figure 7. Relative energy consumption of the register files. The results give the register related energy consumption of a 16-register processor relative to that of a 32-register processor.

In summary we consider an example of a media application mix. In particular, we would like to concentrate on a business videophone application. The application mix consists of an Mpeg decoder for video, Gsm encoder for voice transmission over telco lines, and Rasta speech recognition for minutes of meeting purposes. We weigh the applications as follows: Mpeg 60%, Gsm 20%, and Rasta 20%. We normalize the energy-delay products against the baseline monolithic cache case, and weigh them accordingly for a combined energy-delay product. See Figure 8 for the re-

sults. Minicaches of 512 bytes and 256 bytes give the best results for the L1 data cache only and with L2 cache, respectively. In both cases, we get 30% savings. Ideally, one should also take into account the interplay for multiple applications existing in the same system: issues such as context switches or L2 issues come to mind. We note those issues in the next Section.



(a) Without L2 Cache



(b) With 256K L2 Cache

Figure 8. Relative energy savings for the videophone application for 64K L1 data cache. The minicache energy-delay product is normalized with respect to the monolithic cache.

6. Conclusions

We introduced our Minimax Cache framework for media processors. This framework further blurs the architecture/compiler interface. We discussed the architectural/compiler implementation issues of our approach. We demonstrated that statically mapping frequently used low footprint data such as the scalars in media applications to a separate cache partition is from 30% to 60% energy-delay efficient.

In the future, we would like to extend our analysis to multiple applications residing on the same media processor. This requires an extension to Wattch so that multiple processes and context-switches could be simulated.

In case of the instruction cache, one idea would be to map frequently accessed basic blocks from the secondary task, or interrupt handlers, into the minicache. This would reduce the interference between primary and secondary tasks in multimedia systems, that is a significant source of execution time variability [11]. Alternatively, if frequently accessed basic blocks are mapped, then energy savings could be obtained, similar to the Bellas et al. approach [3].

References

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

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

[3] Bellas N. E., Hajj I. N., Polychronopoulos C. D., "Using Dynamic Cache Management Techniques to Reduce Energy in General Purpose Processors", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 8, No. 6, December 2000

[4] 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

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

[6] 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

[7] 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.

[8] 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

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

[10] Kin J., Gupta M., Mangione-Smith W. H., “The Filter Cache: An Energy Efficient Memory Structure,” *Proceeedings of the 30th Annual Symposium on Microarchitecture, MICRO’97*, 1997

[11] Koopman P.J.Jr.,“Perils of the PC Cache,” *Embedded Systems Programming*, 6(5), May 1993

[12] 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

[13] Lee C., Potkonjak M., Mangione-Smith W. H., “Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems,” *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997

[14] 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

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

[16] 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

[17] Montanaro J., Witek R. T., et al., “A 160MHz 32b 0.5W CMOS RISC Microprocessor,” *IEEE International Solid-State Circuits Conference*, February 1994, vol. 39

[18] 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

[19] 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

[20] 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

[21] Poff D. E., Banikazemi M., Saccone R., Franke H., Abali B., Smith T. B., “Performance of Memory Expansion Technology,” *Workshop on Memory Performance Issues, 28th International Symposium on Computer Architecture, ISCA’01*, Goteborg, Sweden, June 2001

[22] 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

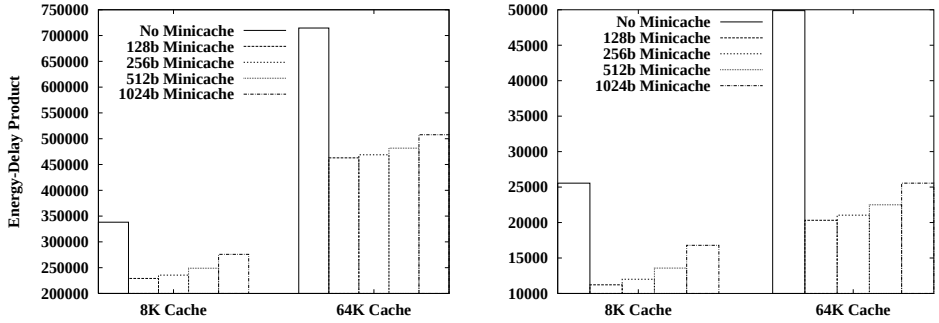
[23] Sanchez F. J., Gonzalez A., Valero M., “Static Locality Analysis for Cache Management,” *Proceedings of the 1997 Conference on Parallel Architectures and Compilation Techniques PACT’97*, 1997

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

[25] 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

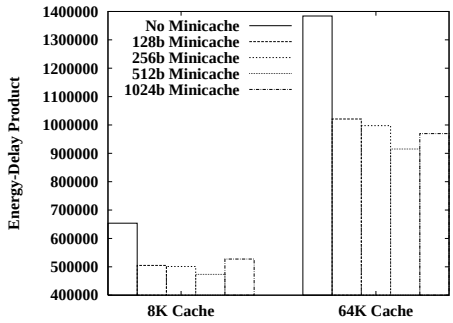
[26] Unsal O. S., Koren I., Krishna C. M., “Power-Aware Replication of Data Structures in Distributed Embedded Real-Time Systems,” *Lecture Notes in Computer Science, LNCS2000 Springer-Verlag*, May 2000

[27] 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

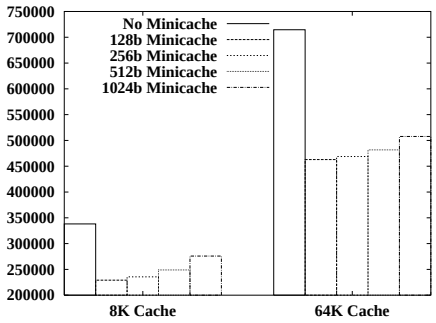


(a) Adpcm

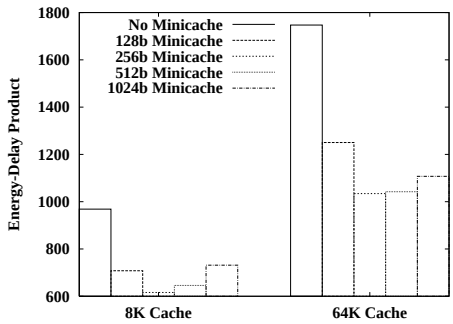
(b) Epic



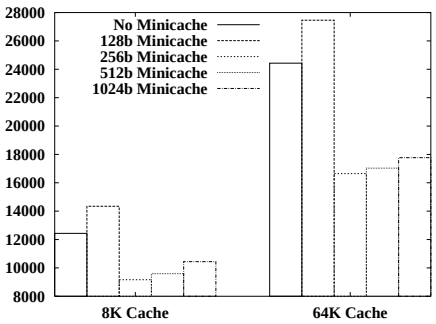
(c) G721



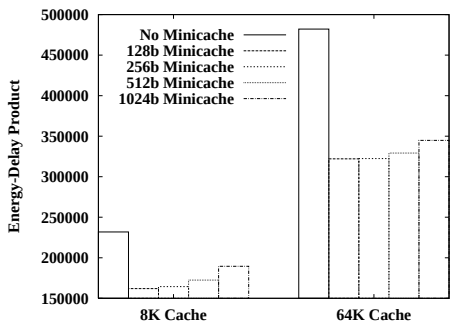
(d) Gsm



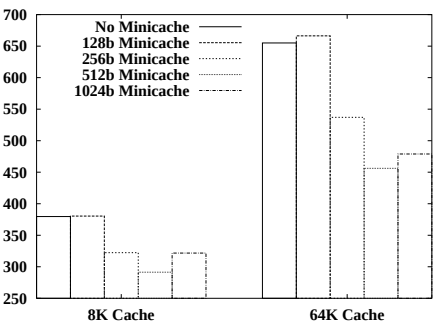
(e) Jpeg



(f) Mesa

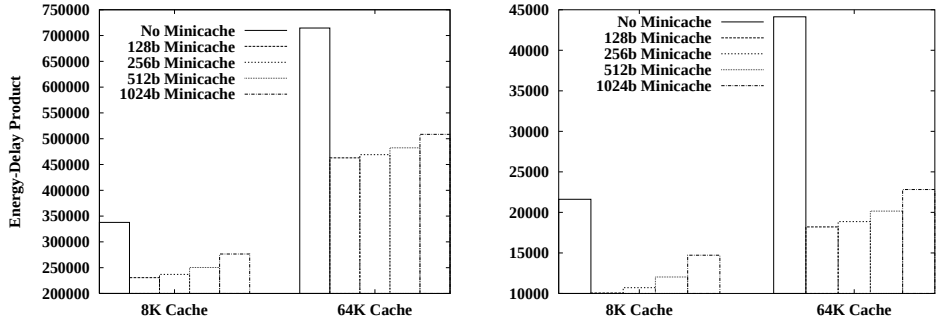


(g) Mpeg



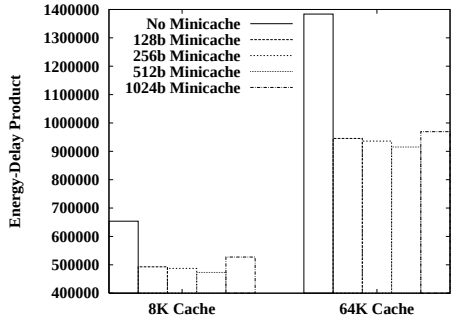
(h) Rasta

Figure 5. L1 data cache Energy-Delay product results for the Minimax Cache.

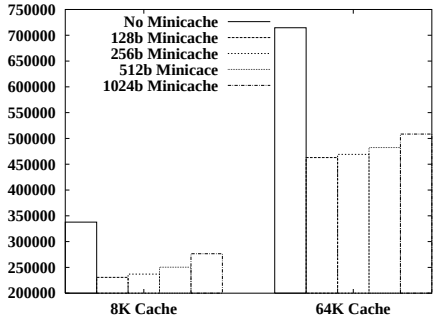


(a) Adpcm

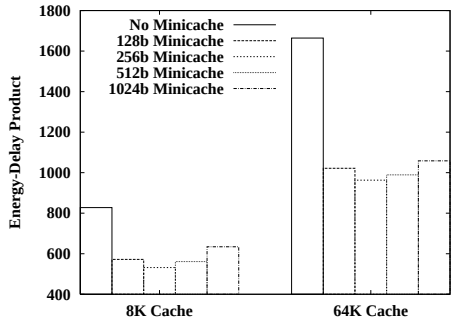
(b) Epic



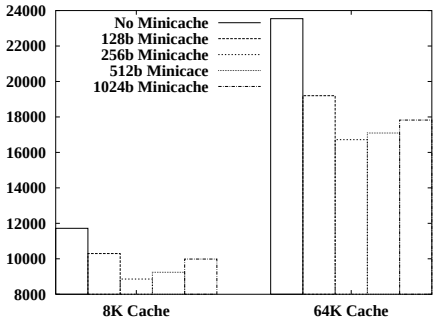
(c) G721



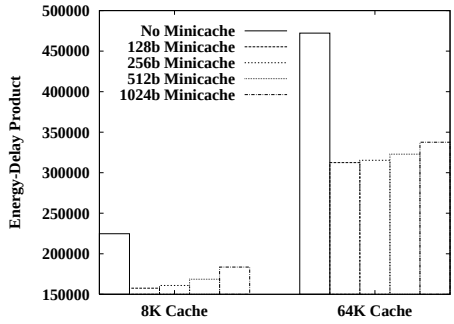
(d) Gsm



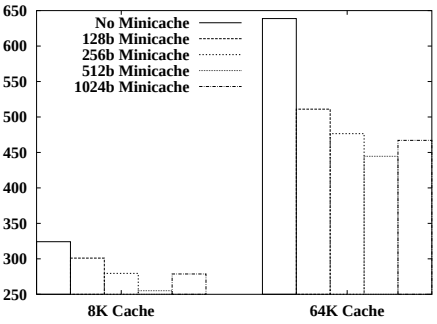
(e) Jpeg



(f) Mesa



(g) Mpeg



(h) Rasta

Figure 6. L1+L2 Cache Energy-Delay product results for the Minimax Cache.