Compilation Techniques for Energy Reduction in Horizontally Partitioned Cache Architectures *

Aviral Shrivastava, Ilya Issenin, and Nikil Dutt
Center for Embedded Computer Systems
Department of Information and Computer Science
University of California, Irvine
Irvine, CA, USA
aviral@ics.uci.edu, isse@ics.uci.edu, dutt@ics.uci.edu

ABSTRACT

Horizontally partitioned data caches are a popular architectural feature in which the processor maintains two or more data caches at the same level of hierarchy. Horizontally partitioned caches help reduce cache pollution and thereby improve performance. Consequently most previous research has focused on exploiting horizontally partitioned data caches to improve performance, and achieve energy reduction only as a byproduct of performance improvement. In constrast, in this paper we show that optimizing for performance trades-off several opportunities for energy reduction. Our experiments on a HP iPAQ h4300-like memory subsystem demonstrate that optimizing for energy consumption results in up to 127% less memory subsystem energy consumption than the performance optimal solution. Furthermore, we show that energy optimal solution incurs on average only 1.7% performance penalty. Therefore, with energy consumption becoming a first class design constraint, there is a need for compilation techniques aimed at energy reduction. To achieve aforementioned energy savings we propose and explore several low-complexity algorithms aimed at reducing the energy consumption and show that very simple greedy heuristics achieve 97% of the possible memory subsystem energy savings.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*; C.0 [**Computer Systems Organization**]: General—*System Architectures*

General Terms

Algorithms, Design, Experimentation, Measurement

*This work was partially funded by grants from Intel Corporation, UC Micro(03-028), and SRC contract 2003-HJ-1111

CASES'05, September 24–27, 2005, San Francisco, California, USA.

Keywords

mini-cache, split cache, horizontally-partitioned cache, energy, compiler, data cache, XScale

1. INTRODUCTION

Advances in compiler technology have far from kept pace with the phenomenal pace of microarchitectural innovation. The (in)famous Proebstring law (in contrast with the Moore's law) states that the compiler advances double the speed of applications every 18 years. Advanced microarchitectural features are employed in processors to be exploited manually and/or with the hope that the compiler will be able to exploit them in the near future. Horizontally partitioned caches are one such feature. Although originally proposed in 1995 by Gonzalez et al. [4], and after being deployed in several current processors such as the popular Intel XScale [8], compiler techniques to exploit it are still in their nascent stages. A horizontally partitioned cache architecture maintains multiple caches at the same level of hierarchy, however each memory address is mapped to exactly one cache. For example, the Intel XScale contains two data caches, a 32KB main cache and a 2KB mini-cache. Each virtual page can be mapped to either of the data caches, depending on the attributes in the page table entry in the data memory management unit. Henceforth in this paper we will call the additional cache as the mini-cache and the original cache as the main cache.

The original idea behind such cache organization is the observation that array accesses in loops often have low temporal locality. Each value of an array is used for a while and then not used for a long time. Such array accesses sweep the cache and evict the existing data (like frequently accessed stack data) out of the cache. The problem is worse for high associativity caches that typically employ FIFO page replacement policy. Mapping such array accesses to the small mini-cache reduces the pollution in the main cache and prevents thrashing, thus leading to performance improvements. Thus a horizontally partitioned cache is a simple, yet powerful architectural feature to improve performance. Consequently most existing approaches for partitioning data between the horizontally partitioned caches aim at improving performance.

In addition to performance improvement, horizontally partitioned caches also result in a reduction in the energy consumption due to two effects. First, reduction in the total number of misses results in reduced energy consumption.

Second, since the size of the mini-cache is typically small, the energy consumed per access in the mini-cache is less than that in the large main cache. Therefore diverting some memory accesses to the mini-cache leads to a decrease in the total energy consumption. Note that the first effect is in-line with the performance goal and was therefore targeted by traditional performance improvement optimizations. However, the second effect is orthogonal to performance improvement. Therefore energy reduction by the second effect was not considered by traditional performance oriented techniques. In fact, as we show in this paper, the second effect (of a smaller mini-cache) can lead to energy improvements even in the presence of slight performance degradation. Note that this is where the goals of performance improvement and energy improvement diverge.

This paper makes several key contributions. First we show that significant savings can be achieved in the memory subsystem energy consumption by optimizing for energy. These savings were not possible while optimizing for performance. Motivated by this, we propose and explore several data partitioning heuristics aimed at reducing the energy consumed by the memory subsystem. Our investigation reveals that very simple greedy heuristics work well for energy optimization. In addition, the energy optimal data partitions incur only a minimal performance loss. Finally we show that our algorithms performs well over various mini-cache associativities and mini-cache sizes.

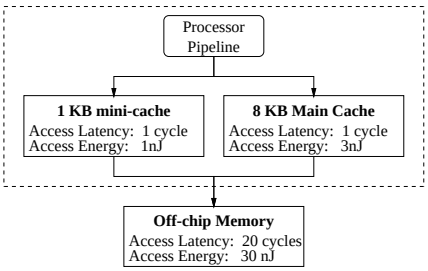
2. RELATED WORK

Caches are one the major contributors of not only system power and performance, but also of the embedded processor area and cost. In the Intel XScale caches comprise approximately 90% of the transistor count, 60% of the area, and consume approximately 15% of the processor power [2]. As a result several hardware, software and cooperative techniques have been proposed to improve the effectiveness of caches.

Originally horizontally partitioned caches were proposed by Gonzalez et al. [4] to separate and thus reduce the interference between the array and stack variables. Most previous research focuses on achieving performance improvements using horizontally partitioned caches. Hardware based approaches improve the performance by partitioning the data based on reuse history. While some approaches use effective address reuse information [15, 11], others use program counter reuse information [17]. Software based approaches attempt to improve performance primarily by coarse-grain region based scheduling [12, 18]. Such schemes simply map the stack data, or the scalar variables to the mini-cache. Recently Xu et al. [19] proposed a profile-based technique to partition the virtual pages to different caches. Their algorithm has complexity $O(m^3n)$, where m is the number of pages accessed in the application, and n is the number of memory accesses. It should be noted that $O(n)$ is also the time complexity of simulation of an application with a given page mapping.

However, the main focus of all the previous research has been performance improvement, and they achieve energy reduction only as a by-product. In this paper we show that optimizing for performance does not effectively optimize energy. With energy consumption becoming an ever important design constraint, there is a need to develop data partitioning techniques aimed at energy improvement. In this paper we propose and evaluate several such data partitioning schemes, and demonstrate that in contrast to high-complexity data partitioning techniques for performance improvements proposed earlier, low-complexity techniques work well for energy optimization with negligible performance penalty.

3. MOTIVATION




```
char a[1024]      // page A
char b[1024]      // page B

for (i=0; i<1024; i++)
   c += a[i] + b[i%5];
```

(b)

Main Cache	Mini Cache	Main Cache Accesses	Main Cache Misses	Mini Cache Accesses	Mini Cache Misses	Runtime (cycles)	Energy (nJ)
A, B	-	2048	65	0	0	3348	8094
A	B	1024	64	1024	1	3348	6046
B	A	1024	1	1024	64	3348	6046
-	A, B	0	0	2048	96	3968	4928

(c)

Figure 1: Motivating Example

We illustrate the difference between optimization for performance and energy for horizontally partitioned caches using the memory architecture shown in Figure 1(a). The example architecture has horizontally partitioned, direct mapped 8KB and 1KB caches, with line size of 16 bytes and other parameters as shown in Figure 1(a). We assume that the page size is 1KB, and that each page can be mapped to either of the caches.

Now we examine the execution of the code in Figure 1(b) on this architecture. The loop accesses two arrays a and b. Assuming that the arrays are aligned to the beginning of the page, they occupy two pages (A and B, one page each). To simplify the analysis, we consider only accesses to these two arrays and evaluate the memory latency (total time spent in memory accesses) and the memory energy consumption.

The table in Figure 1(c) shows the memory latency (in number of cycles) and energy consumption of the memory subsystem for each of the four possible partitions of the pages. When both the pages are mapped to the main 8KB cache, all the 2048 accesses will be to the main cache, and there will be $64 + 1 = 65$ cold misses. We estimate the performance of the memory subsystem as $2048 + 65 \times 20 = 3348$ cycles, and the energy consumed in the memory subsystem as $2048 \times 3 + 65 \times 30 = 8094$ nJ. Similarly we compute the memory latency and memory energy consumption of the other page partitions. The interesting partition is the last one, in which both the arrays are mapped to the small 1KB mini-cache. For this partition, there are more misses in the mini-cache, but less misses in the main cache. The increase in the misses in the mini-cache is more than the decrease in the misses in the main cache. Therefore there is a performance degradation. However the increase in the energy

consumption due to the increased misses in mini-cache is less than the decrease in the energy consumption due to the reduced misses in the main cache, resulting in reduced energy consumption.

Owing to the difference in energy per access between the caches, and the high hit rates of the caches, it intuitively seems that the partition that results in minimum memory energy consumption (optimal energy partition) will have many more pages pages mapped to the mini-cache than the partition that has the least memory latency (optimal performance partition). Thus optimizing for performance is not the same as optimizing for energy. This together with the fact that energy is becoming an ever important design criterion motivates the need for techniques to find optimal energy partitions. In this paper we propose and evaluate several data partitioning algorithms that aim at minimizing memory subsystem energy consumption in horizontally partitioned cache architectures. It turns out that very simple greedy heuristics work well to optimize energy, and furthermore optimal energy partitions do not suffer significant performance degradation.

4. COMPILER FRAMEWORK

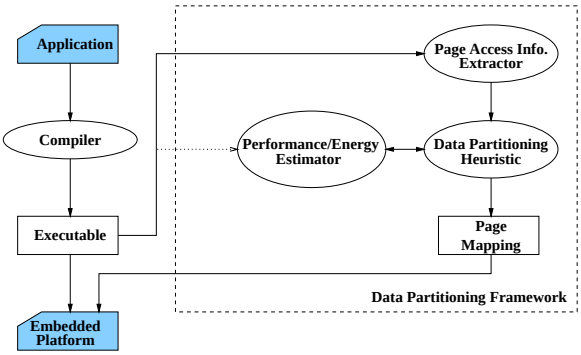


Figure 2: Compiler Framework

The problem of energy optimization for horizontally partitioned caches can be translated into a data partitioning problem. The data memory that the program accesses is divided into pages, and each page can be independently and exclusively mapped to exactly one of the caches. The compiler's job is then to find the mapping of the data memory pages to the caches that leads to minimum energy consumption.

As shown in Figure 2, we first compile the application and generate the executable. The *Page Access Information Extractor* calculates the number of times each page is accessed during the execution of the program. Then it sorts the pages in decreasing order of accesses to the pages. The complexity of simulation used to compute the number of accesses to each page and sorting the pages is $O(n+m\log(m))$, where n is the number of data memory accesses, and m is the number of pages accessed by the application.

The *Data Partitioning Heuristic* finds the best mapping of pages to the caches that minimizes the energy consumption of the target embedded platform. The *Data Partitioning Heuristic* can be tuned to obtain the best performing, or minimal energy data partition by changing the cost function *Performance/Energy Estimator*.

The executable together with the page mapping are then loaded by the operating system of the target platform for optimized execution of the application.

```
Heuristic OMN(Pages P)
01: M = φ, m = φ, U = P
02:  while (U ≠ φ)
03:    p = U.pop()
04:    cost1 = evaluatePartitionCost(M + p, m)
05:    cost2 = evaluatePartitionCost(M, m + p)
06:     if (cost1 ≤ cost2) M+ = p else m+ = p
07: endwhile
08: return M, m
```

Figure 3: Heuristic OMN

5. DATA PARTITIONING ALGORITHMS

The compiler framework depicted in Figure 2 supports using, evaluating, and comparing several data partitioning heuristics of varying complexities to exploit horizontally partitioned caches. All the data partitioning heuristics take as input a list of memory pages accessed by the application, sorted in the order of decreasing accesses to the pages.

The heuristic shown in Figure 3 is a greedy approach for solving the data partitioning problem. Initially, M (list of pages mapped to main cache) and m (list of pages mapped to mini-cache) are empty. All the pages are initially undecided, and are in U (line 01). U is a list containing pages sorted in the decreasing order of accesses. The heuristic picks the first page in U, and evaluates the cost of the partition when the page is mapped to the main cache (line 04) and when it is mapped to the mini-cache (line 05). The heuristic finally maps a page to the partition that results in minimum cost (line 06). Depending on whether the function *evaluatePartitionCost(M, m)* estimates the performance of the partition, or the energy consumption, the heuristic can be used to find the best performing partition, or the minimum energy partition.

evaluatePartitionCost(M, m) uses simulation to estimate the performance or the energy consumption for a given partition. Since each page is considered only once, the complexity of this heuristic is $O(mn)$, where $O(n)$ is the complexity of the simulation-based estimation.

Figure 4 describes a higher complexity heuristic, OM2N. For each page $p \in U$ (line 06), this heuristic uses the heuristic OMN to find whether the page should be mapped to the main cache (lines 04-10) or to the mini-cache (lines 11-17). Therefore the complexity of the heuristic OM2N is $O(m^2n)$.

Our experimental results later show that OMN heuristic works very well for energy reduction. Motivated by this, we developed an even simpler heuristic for data partitioning. Figure 5 shows a very simple single step heuristic. If we define $k = \frac{mini-cache\_size}{page\_size}$, then the first k pages with the maximum number of accesses are mapped to the mini-cache, and the rest are mapped to the main cache. This partition aims to achieve energy reduction while making sure that there is no performance loss (for high associativity mini-caches). Note that for this heuristic we do not need to sort the list of all the pages. Only k pages with the highest number of accesses are required. If the number of pages is m, then the time complexity of selecting the k pages with highest accesses is $O(km)$. Thus the complexity of heuristic is only $O(n + km)$, which can be approximated to $O(n)$, since both k and m are very small as compared to n.

Heuristic OM2N(Pages P)

```
01: M = φ, m = φ, U = P
02: while (U ≠ φ)
03:    p = U.pop()
04:    M1 = M + p, m1 = m, U1 = U
05:    while (U1 ≠ φ)
06:       p' = U1.pop()
07:       cost1' = evaluatePartitionCost(M1 + p', m1)
08:       cost2' = evaluatePartitionCost(M1, m1 + p')
09:       if (cost1' ≤ cost2') M1+ = p' else m1+ = p'
10:    endwhile
11:    M2 = M, m2 = m + p, U2 = U
12:    while (U2 ≠ φ)
13:       p' = U2.pop()
14:       cost1' = evaluatePartitionCost(M2 + p', m2)
15:       cost2' = evaluatePartitionCost(M2, m2 + p')
16:       if (cost1' ≤ cost2') M2+ = p' else m2+ = p'
17:    endwhile
18:    cost1 = evaluatePartitionCost(M1, m1)
19:    cost2 = evaluatePartitionCost(M2, m2)
20:    if (cost1 ≤ cost2) M+ = p else m+ = p
21: endwhile
22: return M, m
```

Figure 4: Heuristic OM2N

Heuristic ON(Pages P)

```
01: M = φ, m = φ
02: for (i = 0; i < mini−cache_size / page_size; i++)
03:    m+ = U.pop()
05: endFor
06: M = U
07: return M, m
```

Figure 5: Heuristic ON

6. EXPERIMENTAL FRAMEWORK

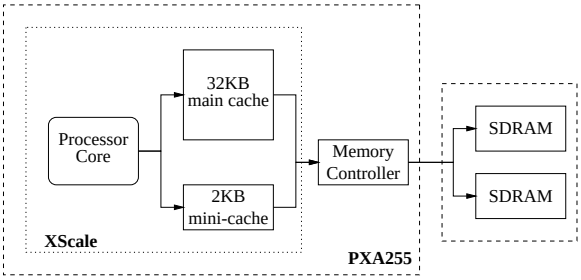


Figure 6: Modeled memory subsystem

We have developed a framework that employs data partitioning algorithms described in Section 5 to optimize the memory latency or the memory subsystem energy consumption of applications. We have modified *sim-safe* simulator from SimpleScalar toolset [1] to obtain the number of accesses to each data memory page. This implements our *Page Access Information Extractor* in Figure 2. In order to estimate the performance/energy of an application for a given mapping of data memory pages to the main cache and the mini-cache, we have developed performance and energy models of memory subsystem of a popular PDA, HP iPAQ h4300 [6].

Figure 6 shows the memory subsystem of the iPAQ that we have modeled. The iPAQ uses the Intel PXA255 processor [7] with the XScale core [8], which has a 32KB main cache and 2KB mini-cache. PXA255 also has an on-chip memory controller that communicates with PC100 compatible SDRAMs via off-chip bus. We have modeled the low-power 32MB Micron MT48V8M32LF [14] SDRAM as the off-chip memory. Since the iPAQ has 64MB of memory, we have modeled two SDRAMs.

Input pin capacitance	3.5 pF
Input/output pin capacitance	5 pF
Bus wire length	2.6 in
PCB characterisitc Impedance, Z_o	60 Ω
Relative permitivity of PCB dielectric, ϵ_r	4.4
Capacitance per unit length, C_o	2.34 pF/in
Capacitance per trace	6.17 pF
Bus energy per burst	9.46 nJ

Table 1: External Memory Bus Parameters

SDRAM current I_{dd}	100 mA
SDRAM supply voltage V_{dd}	2.5 V
Memory bus frequency f_{mem}	100 MHz
Number of memory cycles/burst N_{cyc}	13
SDRAM energy per read/write burst E_{mbst}	32.5 nJ

Table 2: SDRAM Energy Parameters

We use the memory latency as the performance metric. We estimate the memory latency as $(A_m + A_M) + MP \times (M_m + M_M)$, where A_m and A_M are the number of accesses, and M_m and M_M are the number of misses in the mini-cache and the main cache respectively. We obtain these numbers using *sim-cache* simulator [1], modified to model horizontally partitioned caches. The miss penalty MP was estimated as 25 processor cycles, taking into account the processor frequency (400 MHz), memory bus frequency (100 MHz), the SDRAM access latency in power-down mode (6 memory cycles), and the memory controller delay (1 processor cycle).

We use the memory subsystem energy consumption as the energy metric. There are three components in our estimate of memory energy consumption: energy consumed by the caches, energy consumed by off-chip busses, and the energy consumed by the main memory (SDRAMs). We compute the energy consumed in the caches using the access and miss statistics from the modified sim-cache results. The energy consumed per access for each of the caches is computed using eCACTI [13]. As compared to CACTI [16], eCACTI provides better energy estimates for high associativity caches, since it models sense-amps more accurately and scales device widths according to the capacitive loads. We have used linear extrapolation on cache size to estimate energy consumption of the mini-cache, since both CACTI and eCACTI do not model caches with less than 8 sets.

We use the PCB and layout recommendations of PXA255 and Intel 440MX chipset [9] and the relation between Z_o, C_o and ϵ_r [10], to compute the the energy consumed by the external memory bus in a read/write burst as shown in Table 1.

We used the parameters shown in Table 2 from the MICRON MT48V8M32LF SDRAM to compute the energy consumed by the SDRAM per read/write burst operation (cache line read/write), also shown in Table 2.

We perform our experiments on applications from MiBench suite [5] and an implementation of H.263 encoder[3]. To compile our benchmarks we used GCC with all optimizations turned on.

7. EXPERIMENTS

7.1 Optimizing for Energy is different than optimizing for Performance

Our first experiment investigates the difference in optimizing for energy and optimizing for performance on the memory subsystem described in Section 6. We find the partition that results in the least memory latency, and the partition that results in the least energy consumption. Figure 7(a) plots $\frac{E_{br}-E_{be}}{E_{be}}$, where E_{br} is the memory subsystem energy consumption of the partition that results in least memory latency, and E_{be} is the memory subsystem energy consumption by the partition that results in least memory subsystem energy consumption. For the first five benchmarks (*susan - gsm_dec*), the number of pages in the footprint were small, so we could explore all the partitions. For the last seven benchmarks (*jpeg - dijkstra*), we took the partition found by the OM2N heuristic as the best partition. As we present later, OM2N gives very close-to-optimal results in the cases when we were able to search optimally. The graph essentially plots the increase in energy if you choose the best performance partition as your design point. The increase in the energy consumption is up to 130% and on average 58% for this set of benchmarks.

Figure 7(b) plots $\frac{R_{be}-R_{br}}{R_{be}}$, where R_{be} is the memory latency (in cycles) of the best energy partition, and R_{br} is the memory latency of the best performing partition. This graph shows the increase in memory latency when you choose the best energy partition, as compared to using the best performance partition. The increase in memory latency is on average 1.7% and 5.8% in the worst case for this set of benchmarks. Thus choosing the best energy partition results in significant energy savings at a minimal loss in performance.

7.2 Simple Greedy Heuristics work well for Energy Optimization

Next we find out the effectiveness of various data partitioning heuristics. Figure 8(a) plots $\frac{E_{base}-E_{min}}{E_{base}}$ for each heuristic, where E_{base} is the energy consumed in the base case, i.e. when all the data is mapped to the main cache. No page is mapped to the mini-cache, and E_{min} is the energy of the minimum energy partition as computed by the heuristic. The rightmost black bars in the first five benchmarks (*susan - gsm_dec*) represent the energy savings achieved by the optimal search. The optimal search can achieve on average 55% savings in the energy consumed by the memory subsystem. It can also be seen that for these benchmarks, heuristic OM2N achieves close-to-optimal (within 2%) results. Thus we did not feel the need to investigate more complex heuristics. Instead we developed low-complexity heuristics aimed at achieving energy savings. The OMN heuristic achieves up to 62% and on average 50% savings in the energy consumed by the memory subsystem. Thus simple greedy heuristics work well to reduce the energy consumption. These heuristics have lower complexity than the O(m^3n) heuristic suggested in [19] for performance improvement.

In practice, such high complexity algorithms are unlikely to be used due to their large runtimes. For example, for the *jpeg* benchmark, the optimal algorithm would take more than 5,000 years, our OM2N heurtistic takes a few days, the OMN heuristic takes a few hours, and the ON heuristic takes a few minutes to find the optimal partition. Clearly there is a need for low complexity, yet effective data partitioning heuristics.

Furthermore as we have noted that the energy optimal partitions incur minimal loss in performance, therefore for several design domains, it makes sense to use the energy optimal partition. Energy optimal partition saves significant energy at minimal cost. Even the single step ON heuristic is also able to achieve up to 57% and on average 35% energy reduction.

Figure 8(b) plots the goodness of the ON and OMN heuristic in obtaining energy reduction. The goodness of an heuristic is defined as energy reduction achieved by the heuristic as compared to the maximum energy reduction that was possible, i.e. $\frac{(E_{Main}-E_{alg})}{(E_{Main}-E_{best})}$, where E_{Main} is the energy consumption when all the pages are mapped to the main cache, E_{alg} is the energy consumption of the best energy partition that the heuristic found and E_{best} is the energy consumption of the best energy partition. For the last seven benchmarks for which we could not perform the optimal search, we assume the partition found by the heuristic OM2N as the best energy partition. The graph shows that OMN heuristic could obtain on average 97% of the possible energy reduction, while ON could achieve on average 64% of the possible energy reduction. It is important to note here that GCC compiler for XScale does not exploit the mini-cache at all. The ON heuristic provides a simple yet effective way to exploit the mini-cache without incurring any performance penalty (for high associativity mini-cache).

7.3 Sensitivity Analysis

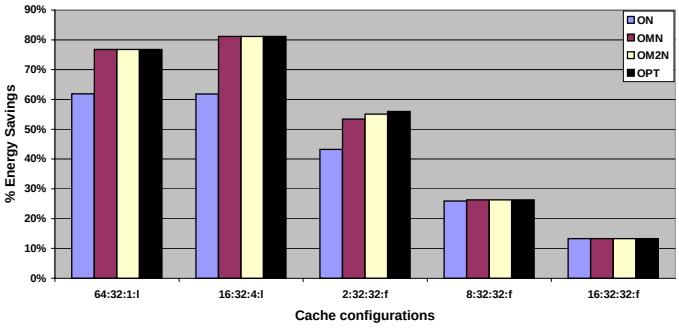
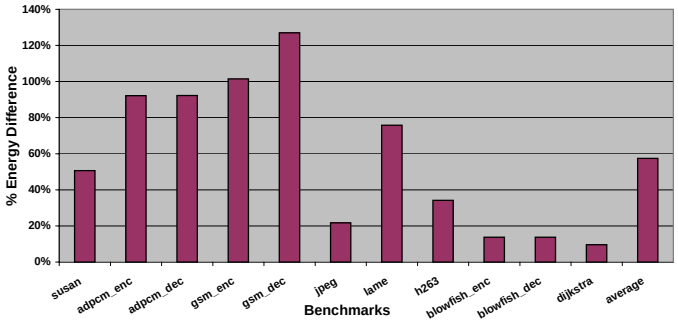


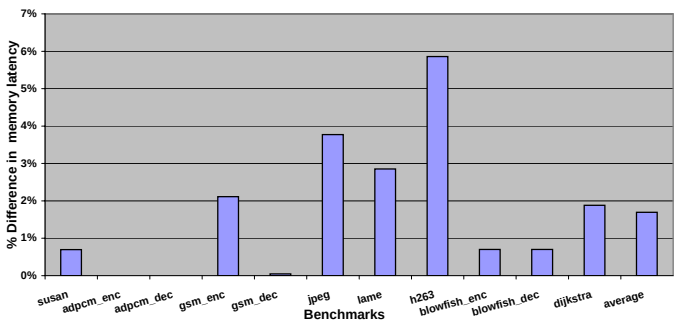
Figure 9: Sensitivity analysis

Next we performed sensitivity analysis of our partitioning algorithms to various cache configurations. Figure 9 plots the average energy savings achieved by the heuristics across several cache configurations on the first five benchmarks. A cache configuration is specified as *no_of_sets : linesize_in_bytes : associativity : replacement_policy*. The first configuration is a direct mapped 2KB cache, while the second configuration is a 4-way set associative 2KB cache. The third configuration is the original configuration, i.e. a 32-way set associative 2KB cache. The fourth configuration is a 32-way, 8KB cache and the fifth configuration corresponds to a 32-way 16KB cache.

In the first two configurations, we vary the associativity of the cache while keeping the size constant, and in the last two configurations, we vary the size of the cache while keeping the associativity constant. For all cache configurations, the difference between the energy consumption of the minimum

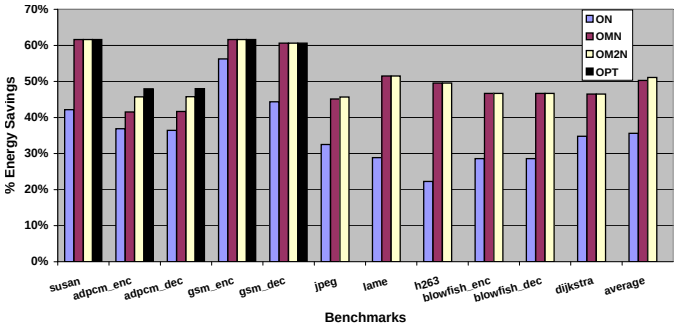


(a) Difference in energy consumption between the best performance partition and the best energy partition

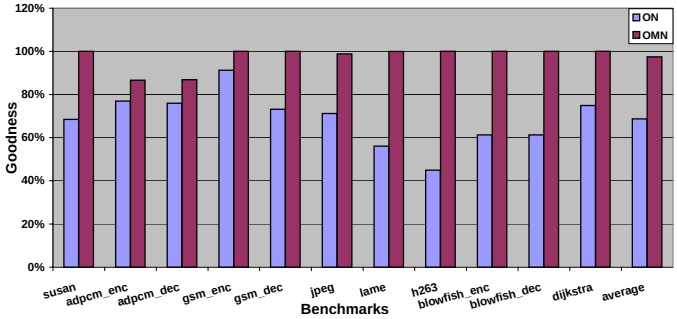


(b) Difference in energy consumption between the best performance partition and the best energy partition

Figure 7: Optimizing for performance is different than optimizing for energy



(a) Energy savings achieved by various heuristics



(b) Goodness of various heuristics

Figure 8: Greedy Heuristics work well for energy Optimization

energy partition found by OM2N, OMN and the optimal algorithm is no more than 2%.

Decreasing the associativity increases the energy reduction, because of the decrease in the energy per access. But for the direct mapped cache, the performance begins to deteriorate, thereby increasing the energy consumption. As we increase the cache size, the energy per access of the mini-cache increases, therefore there is no improvement in the total achievable energy reduction. The best mini-cache configuration for the energy reduction in horizontally partitioned caches for our set of benchmark is the second configuration, i.e. a 4-way set associative 2KB cache.

Thus although the energy gains achieved by changing cache parameters vary, greedy heuristics are able to consistently achieve near optimal results.

8. SUMMARY

Horizontally partitioned caches are a simple yet powerful architectural feature popular in modern embedded processors to improve the performance and energy consumption. Existing compilation techniques to exploit the mini-cache are complex and concentrate on performance improvements. Our experimental results on a HP iPAQ h4300-like memory subsystem show that memory subsystem energy reductions of up to 130% can be achieved by optimizing for energy. The fact that optimizing for energy results in only a minimal 1.7% performance loss (on average) further motivates the need of optimizations aimed at energy reduction. We proposed and evaluated several data partitioning heuristics aimed at energy reduction. We showed that simple greedy heuristics are effective and achieve up to 97% of the possible energy reduction, and that this trend extends to various cache configurations.

Our future work includes exploring the effectiveness of data partitioning heuristics over a wider range of memory architecture parameters.

9. REFERENCES

[1] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[2] L. T. Clark, E. J. Hoffman, M. Biyani, Y. Liao, S. Strazdus, M. Morrow, K. E. Velarde, and M. A. Yarch. An embedded 32-b microprocessor core for low-power and high-performance applications. *IEEE Journal of Solid State Circuits*, 36(11):1599–1608, 2001.

[3] K. O. Lillevold et al. *H.263 test model simulation software.* Telenor R&D, 1995.

[4] A. Gonzalez, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 338–347, New York, NY, USA, 1995. ACM Press.

[5] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE Workshop in workload characterization*, 2001.

[6] Hewlett Packard, http://www..hp.com. *HP iPAQ h4000 Series - System Specifications.*

[7] Intel Corporation, http://www.intel.com/design/pca/applicationsprocessors/manuals/278693.htm. *Intel PXA255 Processor: Developer's Manual.*

[8] Intel Corporation, http://www.intel.com/design/intelxscale/273473.htm. *Intel XScale(R) Core: Developer's Manual.*

[9] Intel Corporation, http://www.intel.com/design/mobile/desguide/251012.htm. *LV/ULV Mobile Intel Pentium III Processor-M and LV/ULV Mobile Intel Celeron*

Processor (0.13u) /Intel 440MX Chipset: Platform Design Guide, 2002.

[10] *IPC-D-317A Design Guidelines for Electronic Packaging Utilizing High-Speed Techniques*, 1995.

[11] T. L. Johnson and W. mei W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *ISCA*, pages 315–326, 1997.

[12] H.-H. S. Lee and G. S. Tyson. Region-based caching: an energy-delay efficient memory architecture for embedded processors. In *CASES '00: Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 120–127, New York, NY, USA, 2000. ACM Press.

[13] M. Mamidipaka and N. Dutt. eCACTI: An enhanced power estimation model for on-chip caches. In *Technical Report TR-04-28, CECS, UCI*, 2004.

[14] Micron Technology Inc., http://www.micron.com/products/dram/mobilesdram/. *MICRON Mobile SDRAM MT48V8M32LF Datasheet*, 2005.

[15] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens. Utilizing reuse information in data cache management. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 449–456, New York, NY, USA, 1998. ACM Press.

[16] P. Shivakumar and N. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. In *WRL Technical Report 2001/2*, 2001.

[17] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture*, pages 93–103, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[18] O. S. Unsal, I. Koren, C. M. Krishna, and C. A. Moritz. The minimax cache: An energy-efficient framework for media processors. In *HPCA '02: Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, page 131, Washington, DC, USA, 2002. IEEE Computer Society.

[19] R. Xu and Z. Li. Using cache mapping to improve memory performance of handheld devices. *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on - ISPASS*, pages 106–114, 2004.