Fast Configurable-Cache Tuning with a Unified Second-Level Cache

Ann Gordon-Ross, Frank Vahid*
Department of Computer Science and Engineering
University of California, Riverside
{ann/vahid}@cs.ucr.edu
http://www.cs.ucr.edu/~vahid
*Also with the Center for Embedded Computer Systems at UC Irvine

Nikil Dutt
Center for Embedded Computer Systems
School of Information and Computer Science
University of California, Irvine
dutt@cecs.uci.edu
http://www.ics.uci.edu/~dutt

ABSTRACT
Tuning a configurable cache subsystem to an application can greatly reduce memory hierarchy energy consumption. Previous tuning methods use a level one configurable cache only, or a second level with separate instruction and data configurable caches. We instead use a commercially-common unified second level, a seemingly minor difference that actually expands the configuration space from 500 to about 20,000. We develop additive way tuning for tuning a cache subsystem with this large space, yielding 62% energy savings and 35% performance improvements over a non-configurable cache, greatly outperforming an extension of a previous method.

Categories and Subject Descriptors
B.3.2 [**Hardware**]: Memory Structures: Design Styles – *cache memories*.

General Terms
Design.

Keywords
Configurable cache, cache hierarchy, cache exploration, cache optimization, low power, low energy, architecture tuning, embedded systems.

1. INTRODUCTION AND MOTIVATION
The memory hierarchy of a microprocessor can consume as much as 50% of the system power in a microprocessor [6]. Such a large contributor to total system power is a good candidate for optimizations to reduce total system power and energy.

Applications require highly diverse cache configurations for optimal energy consumption in the memory hierarchy. Even different phases of the same application may benefit from different cache configurations in each phase [10].

Recent technologies have enabled the tuning of cache parameters to the needs of an application. Core-based processor technologies allow a designer to design a specific cache configuration. Additionally, processor designs with configurable caches are available that can have their caches configured during system reset or even during runtime [6]. Such configurable caches have been shown to have very little size or performance overhead compared to non-configurable caches [6][12].

With the option of cache configuration readily available, a problem is to determine the best cache configuration for a particular application. Previous methods use cache hierarchies with limited configurability, yielding cache configuration spaces of at most a few hundred possible cache configurations, making fast exploration relatively straightforward. Most such methods configure total size, line size, and associativity for only a single level of cache, having less than 50 possible configurations, achieving memory hierarchy energy savings of 40% [12]. A few methods also include a second level of *separate* instruction and data configurable caches, having a few hundred possible configurations, achieving increased memory hierarchy energy savings of 53% [4]. The increased savings suggest that the larger cache configuration space reveals a greater opportunity for energy savings, by allowing the cache to be tuned more closely to an application's needs. However, a larger configuration space makes exploration heuristic development more difficult.

Two-level caches are common in desktop systems and are becoming common in increasingly capable embedded systems. However, the second level cache is commonly *unified* (having one cache with both instructions and data), rather than separate (having one cache for instructions and another for data). A multi-way unified cache enables tradeoffs between the number of instruction ways and the number of data ways, with those tradeoffs known as *way management* [6]. Each way may be used for instructions only, data only, or both instructions and data (or may even be shut down). The interdependence between instruction and data has a (perhaps surprisingly) large impact on the cache configuration space that we must explore. With separated level two caches, we can effectively explore the instruction cache hierarchy independently from the data cache hierarchy, because a configuration of one cache hierarchy doesn't (significantly) affect the other cache hierarchy. In contrast, with a unified second level, the two hierarchies become tightly interdependent, requiring us to consider (roughly) the cross product of the two configuration spaces. For example, two spaces of 200 configurations each, when independent yield 400 configurations to be searched, but when interdependent yield 40,000. Our results will show that this larger space, rather than consisting of uninteresting or impractical configurations, indeed contains useful configurations that allow for intense specialization of the cache hierarchy to an application's needs.

In this paper, we present a heuristic cache tuning method for a highly configurable two-level cache hierarchy. We improve upon previous methods by significantly increasing the search space via a unified second level configurable cache, resulting in greater energy savings than previous methods and increased applicability to current and future systems. Our cache hierarchy allows for 18,144 possible cache

ISLPED05, August 8-10, 2005, San Diego, California USA.

configurations. Our heuristic achieves an average energy savings of 62%, while requiring explicit examination of a mere 0.2% of the search space on average – approximately 34 configurations.

2. RELATED WORK

Commercial systems with tunable caches (e.g., [6]) do not address how to tune those caches, leaving the task to the designer. Several research efforts therefore focus on providing automated assistance for such tuning. Most such efforts focus on single level cache tuning. Platune [3] is a framework for tuning configurable system-on-a-chip (SOC) platforms. Platune offers many configurable parameters and prunes the search space by isolating interdependent parameters from independent parameters. However, the level one cache parameters, being interdependent, are explored exhaustively. Whereas exhaustive exploration was feasible for a level one cache due to the small number of possible configurations, the exhaustive method is not feasible with a highly configurable cache. An exhaustive search of tens of thousands of configurations could take months or more to fully explore.

To speed up exploration time, heuristic methods have been developed. Palesi et al. [7] designed an extension to the Platune tuning environment that used a genetic algorithm to speed up exploration time and produce comparable results. Zhang et al. [12] presents a heuristic method for tuning a configurable cache that searches the cache parameters in their order of impact on energy consumption. The heuristic produces a set of Pareto-optimal points trading off energy consumption and performance.

A few methods exist for tuning two levels of cache, using reduced configurability to maintain a manageable search space. Balasubramonian et al. [1] proposes a method for redistributing the cache size between the level one and level two caches or between the level two and level three caches while maintaining a conventional level one cache. In previous work [4], we designed an exploration heuristic for a configurable cache hierarchy that explores separate level one instruction and data caches and separate level two instruction and data caches.

3. CONFIGURABLE CACHE ARCHITECTURE

Our configurable two-level cache architecture consists of separate configurable level one caches and a unified level two cache. The level one configurable cache architecture is based on the tunable cache described in [13]. Hardware layout verification for the configurable cache is presented in [12]. The tunable parameters consist of cache size, line size, and associativity. The base cache structure in an 8 KB cache consisting of four 2 KB banks where each bank acts as a way. Special way configuration registers allow for a 2-way set associative and a direct mapped cache using way concatenation. Additionally, ways may be shut down to allow for direct mapped and 2-way set associative 4 KB caches and a direct mapped 2 KB cache. Way shutdown and way concatenation can be combined to allow a direct-mapped 4 KB cache.

The second level cache is a configurable unified cache quite different than the first level cache. For the second level, we utilize way management implemented in Motorola's M*CORE processor [6]. Way management allows for each particular way in a unified cache to be specified as a unified way (instruction and data), an instruction-only way, a data-only way, or the way can be shut down entirely.

For the exploration parameters, we chose values to reflect typical off-the-shelf embedded systems. For the level one cache, we explore 2, 4, and 8 KB cache sizes, 16, 32, and 64 byte line sizes, and direct-mapped, 2-, and 4-way set associativities. For the level two cache, we use a 64 KB cache with four configurable ways and configurable line sizes of 16, 32, and 64 bytes. However, our heuristic is not dependent on these values, nor on embedded applications – for desktop applications, larger total-size values would be appropriate. For comparison purposes, we use a common cache configuration to act as a base cache configuration to show the effectiveness of our cache tuning heuristic in reducing energy. The base cache configuration is an 8 Kbyte 4-way set associative cache with a 32 byte line size for the level one cache, and a 64 Kbyte fully unified cache with a 64 byte line size for the level two cache – a reasonably common configuration.

Our configurable cache architecture offers 18,144 different configurations. For each level one cache, there are 18 different cache configurations (configurable parameters are size, line size, and associativity, each with three possible values, minus invalid combinations). The second cache level has 36 unique combinations of way configuration for each of the three line sizes, resulting in 108 different level two configurations. Thus, the maximum number of cache configurations is 34,992. However, the second level line size must be equal than or greater than the largest level one line size, reducing the design space to 18,144 – still a very large number of configurations.

4. TUNING HEURISTICS

4.1 Sequential Exploration with Ratio Projection

A simple tuning heuristic for two-level caches ignores the tuning dependency between the level one instruction and data caches, and sequentially explores the two levels, first tuning level one, then level two. As previous tuning methods don't consider a unified cache, we first developed a sequential heuristic for two-level caches, providing a close comparison to current methods, and illustrating the need to fully explore the tuning dependencies.

For level one exploration, our heuristic explores parameters in the order of their impact on the energy consumption, with higher impact parameters explored first [13]. Cache size is explored first followed by line size and then associativity. To reduce cache flushing during exploration, the heuristic explores each parameter starting with the smallest value and increasing to the largest value. For the level two cache, the heuristic must also consider that the cache is unified. Thus, not only must the heuristic determine the total size, line size, and ways, but the heuristic must also determine how many ways will be for data, how many for instruction, how many for both instruction and data, and how many will be shut down. For unified level two cache exploration, we initially developed a method we call ratio projection.

The ***ratio projection*** method projects the number of necessary instruction and data ways needed for the best cache configuration. Ratio projection sets the level two cache to have one instruction way and adds data ways one at a time. The lowest energy configuration suggests the ideal number of data ways needed in the level two cache. The method determines the ideal number of instruction ways similarly. The method then must combine the ideal number of instruction and data ways.

Simply adding the number of ways could exceed the available number of ways in the level two cache. Instead, the method decreases the number data and instruction ways and/or unifies them and trying to keep the *ratio* of instruction to data ways as close to the ideal as possible. For example, the method might determine the ideal number of instruction and data ways to be 3 and 3, respectively. Given only four available ways, the ratio projection method would allocate 2 instruction and 2 data ways, thus maintaining the same ratio of instruction to data ways. Further details of the ratio projection method are available in [5].

The sequential heuristic performed poorly for many benchmarks. Although the heuristic resulted in a 20-40% decrease in energy consumption over the base cache configuration for most examples, poor performance on some benchmarks (as much as 3.6x more energy) resulted in the heuristic yielding an average energy *increase* of 24%. Clearly, a simple adaptation of current methods does not sufficiently explore tuning dependencies.

4.2 Alternating Cache Exploration with Additive Way Tuning – ACE-AWT

The poor results of the first heuristic substantiate the hypothesis that precise exploration with regards to tuning dependencies is necessary. Exploring the level one cache separately from the level two cache naively ignores the dependency that exists between the two levels via the level two unified cache. For example, altering a parameter in the level one instruction cache changes the effectiveness of the level two cache by changing the quantity of level two fetches and the addresses fetched. Also, the change in level two utilization by instructions affects the level one data cache by changing the contention among instructions and data in the shared level two cache.

In [4], we similarly concluded the importance of tuning levels one and two together (though instruction and data were separate in that work), and we thus designed the *interlaced* exploration method. Instead of fully exploring the level one cache and then proceeding to the level two cache, the interlaced method explores one parameter for the level one cache and then for the level two cache, before proceeding to explore the next parameter. However, that interlaced method only addressed dependency between separate level one and level two caches, and not the dependency between the level one instruction and data caches. Additionally, the interlaced method cannot be easily adapted to a unified cache featuring way management.

For level two exploration, way management makes interlaced exploration of the cache levels difficult because of the dependency between size and associativity. To change cache size, way management either adds or removes a way. However, the added or removed way is either a unified, data, or instruction way, thus affecting associativity. Similarly, when changing the cache's associativity, a way is either added or removed, which changes the cache size. This dependency complicates level two cache exploration, preventing exploring either associativity or size alone.

To overcome the difficulty arising in interlaced exploration and to extend the interlaced heuristic to address level one instruction and data cache dependencies, we introduce the alternating cache exploration with additive way tuning heuristic for level two cache exploration (ACE-AWT). For each cache parameter, the ACE-AWT heuristic first tunes the level one instruction cache, then the level one data cache, followed by additive way tuning for the level two cache. The first phase of additive way tuning, illustrated in Figure 1(a), adds ways one at a time and chooses the next way to add based on what type of added way resulted in the lowest energy cache configuration. Additive way tuning starts by adding one way to the level two cache, and then explores three configurations – a single instruction, data, or unified way. The heuristic chooses the lowest-energy configuration, and then adds another way to the level two cache, again trying an instruction, data, or unified way. This additive method of increasing the cache size and associativity continues until the level two cache is full or until there is no longer a decrease in energy consumption. This phase of additive way tuning is done when the level two cache size is initially explored.

Alternating level exploration with a unified second level of cache increases the difficulty of exploring the line size. The line size of the level two cache must always be equal or greater than the line sizes of both of the level one instruction and data caches. To allow for level one line size exploration, our heuristic increases the size of the level two line size while increasing the size of the level one line size. After determining level one line sizes, the ACE-AWT heuristic explores remaining larger level two line sizes.

During associativity exploration, Figure 1(b) illustrates the final tuning step applied to fine tune the cache configuration. The ACE-AWT heuristic adjusts ways to hone in on the best cache configuration by attempting to add and/or remove ways. First, the heuristic tries to increase the number of ways by adding either an instruction, data, or unified way one at a time. If the cache size is full, the heuristic skips the enlargement step. The heuristic then explores decreasing the size of the cache by removing an instruction, data, or unified way one at a time. If removing a way causes the cache to be empty, that configuration is ignored. The lowest energy cache configuration is chosen if it improves upon the current cache configuration. This tuning step is continued until there is no improvement in energy consumption or there is no previously unexplored configuration to explore. Further details of the ACE-AWT heuristic are available in [5].

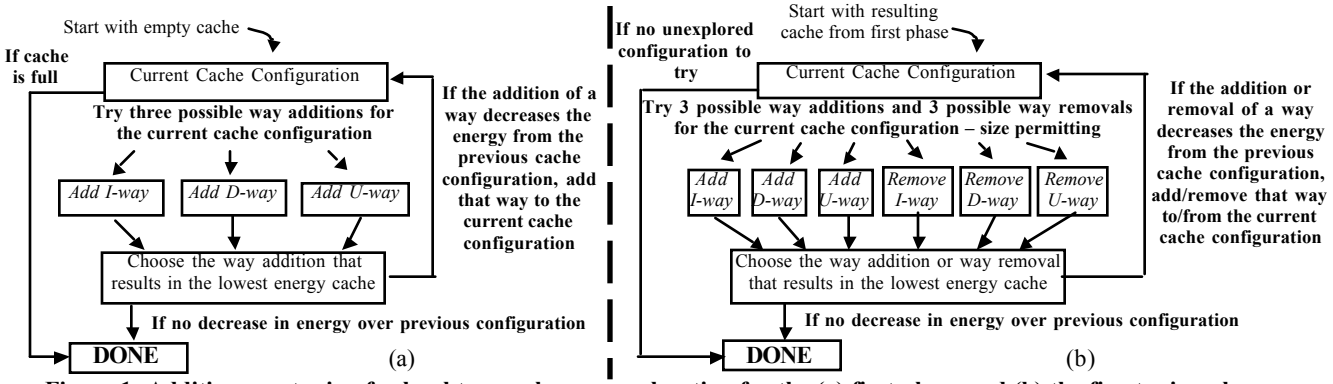

Figure 1: Additive way tuning for level two cache way exploration for the (a) first phase and (b) the fine tuning phase.

5. RESULTS

We applied each heuristic to 27 benchmarks - sixteen benchmarks from the EEMBC benchmark suite [2] and eleven benchmarks from the Powerstone benchmark suite [6]. These benchmarks are all embedded system benchmarks and thus suitable for the configurable cache parameter values we examined. We stress that we could also run desktop benchmarks using suitable cache parameter values, and we are doing so for related and future work.

We used estimation methods and measurements to calculate the total system energy consumption, including both dynamic and static energy. We used CACTI [9] to determine the dynamic energy consumption consumed by a cache fetch for each cache configuration for 0.18-micron technology. We obtained the main memory fetch energy using a standard Samsung memory, and CPU stall energy from a 0.18-micron MIPS microprocessor (details are available in [5]). Our energy numbers represent all memory-access-related energy only. We estimate cache static energy as 10% of total cache energy – a reasonable assumption for current and near future technology. For miss penalties and throughput for both cache levels, we estimate ratios typical for an embedded system. We assume a level two fetch is four times slower than a level one fetch, and a main memory fetch is ten times slower than a level two fetch. We assume memory throughput is 50% of latency, meaning blocks fetches after the first block take 50% of the latency of the first block fetch.

We modified SimpleScalar to simulate way management in the level two cache and to determine cache hit and miss values for each cache configuration. We ran exploration scripts that applied each heuristic to every benchmark.

Figure 2 shows the energy consumption for each benchmark for both tuning heuristics, and shows the optimal cache energy consumption for 12 randomly chosen benchmarks (we couldn't generate optimal energy for every benchmark due to the large time required). Energy consumption for each heuristic is normalized to the energy consumption of the base cache for each benchmark. Figure 2 shows that while the sequential with ratio projection heuristic performed well on a number of benchmarks, the average energy *increased* due to poor heuristic performance on several benchmarks. However, the ACE-AWT heuristic achieves energy savings for every benchmark, resulting in an average 62% energy savings. For the benchmarks with optimal cache configuration information, the ACE-AWT either finds the optimal or near-optimal configuration. The ACE-AWT achieves these energy savings by exploring only 34 unique configurations on average over all benchmarks – a mere 0.2% of the total search space.

We also examined the performance impact of the ACE-AWT heuristic. In real time systems, negative performance impacts are likely unacceptable. We observed that for the ACE-AWT heuristic, each benchmark shows an *improvement* in performance with an average speedup of 35%. We found that this improvement comes due to tuning the line size to the locality needs of the application [11]. (While this result may seem surprising, the compromise line size found in most caches, typically 32 bytes, may perform best on average across all benchmarks, but specific applications often do much better with either a 16 byte or 64 byte line size).

6. CONCLUSIONS AND FUTURE WORK

We presented a new heuristic for tuning a two-level cache with a unified second level, yielding an average 62% memory-access-energy savings over a base cache configuration, while exploring only 0.2% of the design space. Future work includes applying our heuristic to desktop/server applications and architectures. We are also investigating finer-grained cache tuning to program phases.

7. ACKNOWLEDGEMENTS

This research was supported in part by the National Science Foundation (CCR-0203829, CCR-9876006) and by the Semiconductor Research Corporation.

8. REFERENCES

[1] Balasubramonian, R., Albonesi, D., Buyuktosunoglu, A., Dwarkadas, S. Memory heirarchy reconfiguration for energy and performance in general-purpose processor architecture. 33rd International Symposium on Microarchitecture, December 2000.

[2] EEMBC, the Embedded Microprocessor Benchmark Consortium, www.eembc.org.

[3] Givargis, T., Vahid, F. Platune: a tuning framework for system-on-a-chip platforms. IEEE Transactions on Computer Aided Design, November 2002.

[4] Gordon-Ross, A., Vahid, F., Dutt, N. Automatic tuning of two-level caches to embedded applications. Design, Automation and Test Conference in Europe (DATE), 2004.

[5] Gordon-Ross, A., Vahid, F., Dutt, N., Fast configurable-cache tuning with a unified second-level cache. UC Riverside Technical Report UCR-CS-2005-05002.

[6] Malik, A., Moyer, W., Cermak, D. A low power unified cache architecture providing power and performance flexibility. International Symposium on Low Power Electronics and Design, 2000.

[7] Palesi, M., Givargis, T. Multi-objective design space exploration using genetic algorithms. International Workshop on Hardware/Software Codesign, May 2002.

[8] Personal communication with M*CORE designers

[9] Reinman, G., Jouppi, N.P. Cacti2.0: an integraded cache timing and power model. COMPAQ Western Research Lab, 1999.

[10] Sherwood, T., Perelman, E., Hamerly, G., Sair, S., Calder, B. Discovering and Exploiting Program Phases, IEEE Micro: Micro's Top Picks from Computer Architecture Conferences, December 2003

[11] Veidenbaum, A., Tang, W., Gupta, R., Nicolau, A., Ji, X. Adapting cache line size to application behavior. International Conference on Supercomputing, June 1999.

[12] Zhang, C., Vahid, F., Najjar, W. A highly-configurable cache architecture for embedded systems. 30th Annual International Symposium on Computer Architecture, June 2003.

[13] Zhang, C., Vahid, F. A self-tuning cache architecture for embedded systems. Design, Automation and Test Conference in Europe (DATE), 2004

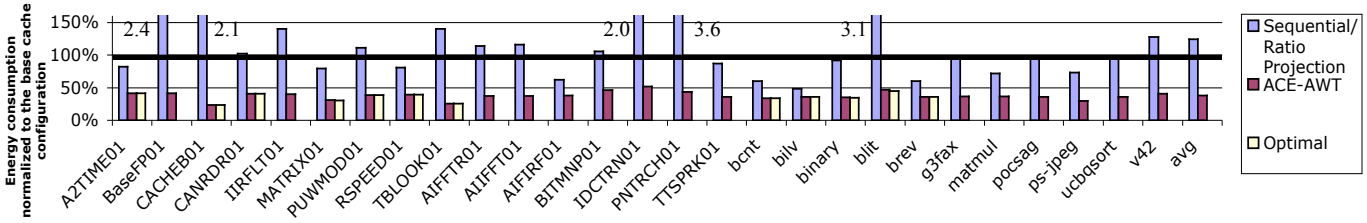

Figure 2: Energy consumption normalized to the base cache configuration (bold line) for both cache exploration heuristics and the optimal cache configuration.