

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

## 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.*

## 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 [11][17]. Such a large contributor to total system power is a good candidate for optimizations to reduce total system power and energy. Low power or energy is needed not only in embedded systems that run on batteries or have limited cooling ability, but also in desktop and mainframes where chips are requiring costly cooling methods.

Applications require highly diverse cache configurations for optimal energy consumption in the memory hierarchy [22]. Even different phases of the same application may benefit from different cache configurations in each phase [12][18]. For example, the size of the cache should reflect the working set of the application. Too large of a cache would result in cache fetches consuming excessively high energy. Too small of a cache would result in wasted energy due to thrashing in the cache, with frequently used items repeatedly swapped in and out of the cache. Additionally, the cache line size and associativity should reflect the needs of a particular application or application phase to achieve the most energy efficient cache configuration.

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 [2][3][4][13][19]. Additionally, processor designs with configurable caches are available that can have their caches configured during system reset or even during runtime [1][11][22]. Such configurable caches have been shown to have very little size or performance overhead compared to non-configurable caches [11][21].

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% [21]. 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% [10]. The increased savings suggest that increasing the size of the 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, increased configuration space leads to increased difficulty in exploration heuristic development.

Two-level caches are common in desktop systems and becoming quite common in increasingly capable embedded systems. However, the second level of such two-level caches 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 [11]. Each way may be used for instructions only, data only, or both instructions and data (or may even be shut down). An example configuration of a four-way unified cache is 3 instruction ways and 1 data way; another example is 2 instruction ways, 1 data way, and one instruction/data way. The interdependence 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.

How to adapt existing cache tuning methods to a way-managed unified second level cache is not obvious, due to the increased tuning interdependency between the caches. Previous methods limited tuning dependency to limit the configuration space, thus making heuristic development easier. Previous tuning methods that address the tuning dependency between the level one and separate level two caches cannot be directly applied to a unified second level of cache.

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 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. We also examine the effects of increasing static energy on the fidelity cache configuration heuristics. We further describe how our cache tuning heuristic is efficient enough to be used in simulation environments, while at the same time being simply enough to be implemented in an on-chip dynamic tuning approach.

## 2. Related Work

Commercial systems with tunable caches (e.g., [4][11]) 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 [8] 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 dependent, 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.

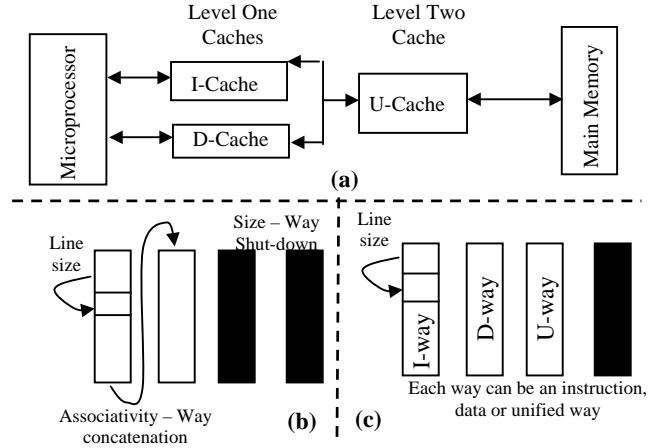
To speed up exploration time, heuristic methods have been developed. Palesi et al. [14] designed an extension to the Platune tuning environment that used a genetic algorithm to speed up exploration time and produce comparable results. Zhang et al. [21] presents a heuristic method for tuning a configurable 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. Ghosh et al. [9] presents a heuristic that, through an analytical model, directly determines the cache configuration based on the designers performance constraints.

A few methods exist for tuning two levels of cache, using reduced configurability to maintain a manageable search space. Balasubramonian et al. [5] proposes a method for redistributing the cache size between the level two and level three caches while maintaining a conventional level one cache. Gordon-Ross et al. [10] designs 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, shown in Figure 1(a), 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 [22] and is illustrated in Figure 1(b). Hardware layout verification for the configurable cache is presented in [21]. 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 a direct mapped and 2-way set associative 4 KB cache and a direct mapped 2 KB cache. As a result of the configurable banks, 2 KB 2-way or 4-way set associative caches and a 4 KB 4-way set associative cache are not possible configurations. This limitation is only applicable to a hardware based configurable cache. In a simulation-based exploration, any cache configuration is possible.

The second level cache is a configurable unified cache quite different than the first level cache, illustrated in Figure 1(c). For the second level, we utilize way management implemented in Motorola’s M\*CORE processor [11]. Way management allows for each particular way in a unified cache to be specified as a



**Figure 1: Configurable Cache Architecture: (a) the cache hierarchy used, (b) configurability available for the level one caches, and (c) configurability available for the level two cache.**

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 consisting of 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.

Our configurable cache architecture offers 18,144 different cache 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). 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, restrictions reduce the number of configurations. As described above, not all associativities are possible for each cache size. Further, the second level line size must be equal than or greater than the largest level one line size. With these restrictions, the design space reduces to 18,144 – still a very large number of configurations.

Due to the huge exploration space, exhaustive exploration to determine the optimal cache configuration for every benchmark for comparison with our heuristic is not feasible as it would take more than a year. Even so, we generated optimal results for 5 randomly chosen benchmarks. For comparison purposes we also 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 consists of 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.

## 4. Tuning Heuristics

For our configurable cache hierarchy, the full configuration space consists of 18,144 different configurations. Even if the time to explore one configuration only took only half a second, exploring all configurations for a benchmark would still take half an hour – clearly not feasible for a dynamic tuning method. If exploring each configuration took five minutes (a typical runtime for a simulation-based tuning approach on contemporary workstations), it would take 63 days to exhaustively explore the search space for a single benchmark. We sought to develop a tuning heuristic to efficiently explore a small portion of the search space and produce good energy savings over the base cache configuration. We considered two possible heuristics, which we now describe.

### 4.1 Sequential Exploration with Ratio Projection

A straightforward heuristic is to ignore the tuning dependency between the level one instruction and data caches and tune the level one caches while ignoring the second level cache. After the first level caches are tuned, the second level of cache is then tuned. State-of-the-art tuning methods are not applicable to a cache with way management so we present this exploration heuristic as a close comparison to current methods and to illustrate the need to fully explore the tuning dependencies.

For level one exploration, the parameters are explored in the order of their impact on the energy consumption with higher impact parameters explored first [22]. Cache size is explored first followed by associativity and then line size. To reduce the amount of necessary cache flushing, each parameter is explored starting with the smallest value and increasing to the largest value. The same heuristic cannot be used directly for the level two cache because that 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 level two cache exploration, we initially designed the ratio projection method of level two cache exploration.

**Ratio projection**, illustrated in Figure 2, 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. Similarly, the ideal number of instruction ways needed for the level two cache is also determined. To determine the level two cache configuration, the

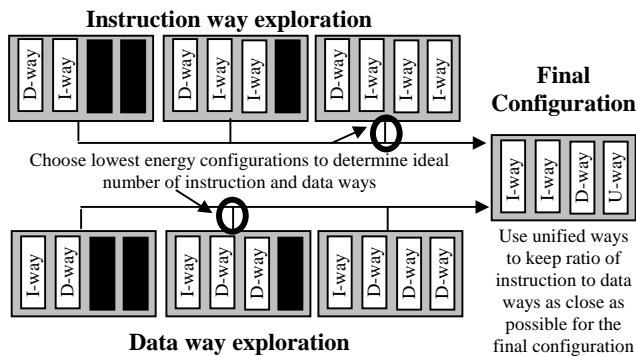


Figure 2: Ratio projection for level two cache way exploration.

ideal number of instruction and data ways are combined. However, simply combining the number of ways could exceed the available number of ways in the level two cache. To account for this situation, we decrease the number data and instruction ways by unifying them and trying to keep the ratio of instruction to data ways as close to the ideal as possible.

We combined the level two tuning method to the level one tuning heuristic and discovered that the heuristic did not perform well for many of the benchmarks. Whereas for the majority of the benchmarks, the tuning method showed a 20-40% decrease in energy consumption over the base cache configuration, the average energy consumption over all benchmarks actually *increased* by 24%. In the worst case, a cache configuration consuming 3.6 times more energy than the base cache was found by the heuristic. 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 exploring the level two cache naively assumes that the tuning of the caches can be done independently. But, tuning dependency exists between the level one instruction and data caches via the level two unified cache, and between the level one and level two caches. 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.

Gordon-Ross [10] arrived at a similar conclusion about level one and level two cache dependencies, and designed the interlaced method of exploration to interlace the tuning of the level one and level two caches. Instead of fully exploring the level one cache and then proceeding to the level two cache, each parameter is explored for the level one cache and then the level two cache before proceeding to explore the next parameter. However, the interlaced method developed by Gordon-Ross only addresses 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 adapted to a 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 exploration. To change the size, either a way is added or removed from the cache. However, the added or removed way is either a unified, data, or instruction way, additionally affecting the associativity. Similarly, when changing the cache’s associativity, a way is either added or removed which also changes the size of the cache as well. This dependency complicates the exploration of the level two cache, since we can’t just explore either associativity or size alone.

To overcome the difficulty arising in interlaced exploration we designed the alternating cache exploration with additive way tuning heuristic for level two cache exploration – or ACE-AWT for short. For each cache parameter, exploration will first tune 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 3(a), adds ways one at a

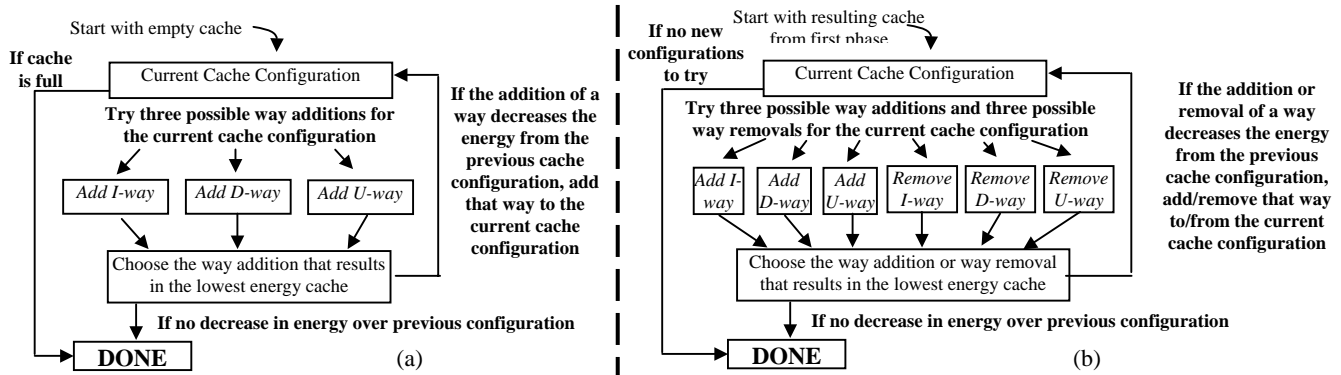


Figure 3: Additive way tuning for level two cache way exploration for the (a) first phase and (b) the fine tuning step.

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 three configurations are explored – a single instruction, data, or unified way. The configuration with the lowest energy consumption is chosen and another way is added to the level two cache, again trying either an instruction, data, or unified way. This additive method of increasing the cache size and associativity is continued until the level two cache is full or 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. It is necessary to make sure that the line size of the level two cache is always 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, it is necessary to increase the size of the level two line size while increasing the size of the level one line size. After the level one line sizes are determined, the remaining larger level two line sizes are explored

.During associativity exploration, Figure 3(b) illustrates the final tuning step applied to fine tune the cache configuration. The ways are adjusted to hone in on the best cache configuration by attempting to add and/or remove ways. First, we try to increase the number of ways by adding either an instruction, data, or unified way one at a time. We then explore decreasing the size of the cache by removing an instruction, data, or unified way one at a time. 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.

## 5. Results

### 5.1 Experimental Setup

We applied each heuristic to 27 benchmarks - sixteen benchmarks from the EEMBC benchmark suite [7] and eleven benchmarks from the Powerstone benchmark suite [11]. 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.

Estimation methods and measurements are used to calculate the total system energy consumption including both dynamic and static energy. We use CACTI [16] 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. 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% the latency of the first block. Gordon-Ross [10] shows that cache hierarchy exploration heuristics maintain good integrity across systems with varying memory latency and throughput.

We modified SimpleScalar [6] 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

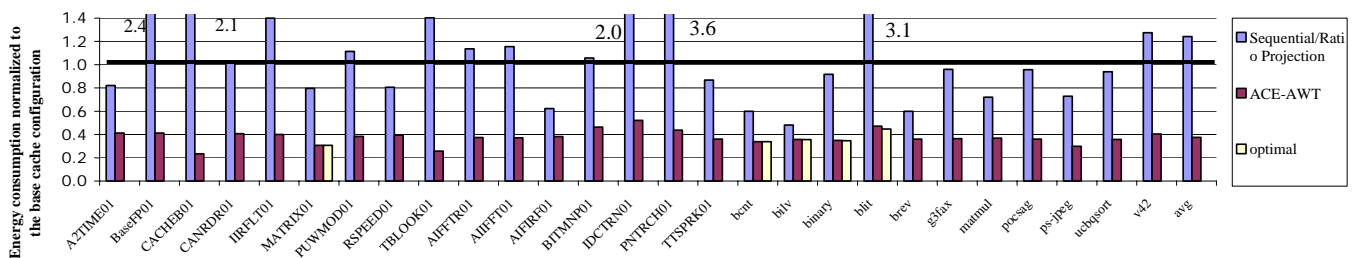
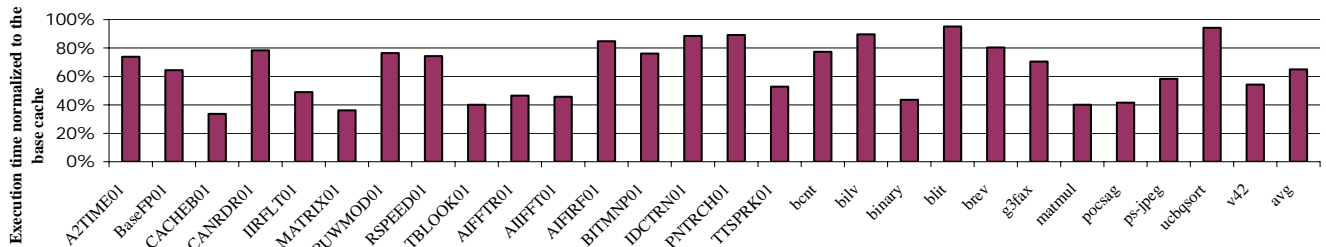


Figure 4: Energy consumption normalized to the base cache configuration (bold line) for both cache exploration heuristics and the optimal cache configuration.



**Figure 5: Execution time of the benchmarks for alternating cache exploration with additive way tuning heuristic (ACE-AWT) normalized to the execution time of the benchmark with the base cache configuration**

applied each heuristic to every benchmark.

## 5.2 Energy Consumption and Performance

Figure 4 shows the energy consumption for each benchmark for both tuning heuristics and the optimal cache energy consumption for 5 randomly chosen benchmarks (we are continuing to generate optimal cache configurations for the remaining benchmarks). Energy consumption for each heuristic is normalized to the energy consumption of the base cache for each benchmark. Figure 4 shows that while the sequential with ratio projection performed well on a number of benchmarks, on average the energy *increased* over all benchmarks with some benchmarks consuming significantly more energy over the base cache configuration. However, the ACE-AWT heuristic improves greatly over the initial heuristic showing energy savings of 62% averaged over all benchmarks. For the 5 benchmarks with optimal cache configuration information, the ACE-AWT either finds the optimal cache configuration or determines a cache configuration that is very near the optimal. The ACE-AWT achieves these energy savings by exploring only 34 unique configurations – a mere 0.2% of the total search space.

As well as showing good energy savings across all benchmarks, the performance impact of the ACE-AWT heuristic should be examined. In real time systems, negative performance impacts are likely unacceptable. Figure 5 shows the execution time of each benchmark for the ACE-AWT heuristic normalized to the execution time for the base cache configuration. 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 configuration to the locality needs of the application [20].

## 5.3 Static Energy for Future Technology

For the results presented in section 0, we assumed static energy accounted for 10% of the total energy consumption of the cache. However, static energy becomes a greater factor in total energy consumption as technology pushes further in deep sub-micron feature sizes, and it is interesting to investigate the fidelity of cache configuration. We explored systems where static energy accounted for 15%, 20%, 25%, and, for possible farther distant

|                   | Sequential/Ratio Projection | ACE-AWT     |
|-------------------|-----------------------------|-------------|
| 10% Static Energy | 1.24                        | <b>0.38</b> |
| 15% Static Energy | 1.18                        | <b>0.37</b> |
| 20% Static Energy | 1.10                        | <b>0.33</b> |
| 25% Static Energy | 1.05                        | <b>0.32</b> |
| 50% Static Energy | <b>0.82</b>                 | <b>0.23</b> |

**Table 1: Energy consumption normalized to the base cache configuration averaged across all benchmarks for different static energy consumption. Energy savings are shown in bold.**

technologies, 50% of the total energy consumption of the cache.

Table 1 shows the average energy consumption normalized to the base cache configuration averaged across all benchmarks for the heuristics studied. Averages with *energy savings* are shown in bold. The ACE-AWT heuristic shows very good fidelity with increasing static energy consumption.

Both heuristics show the same trend – as the percentage of static energy consumption increases, the cache tuning heuristics are revealing greater energy savings. This trend is expected since cache tuning improves performance and thus eliminates costly idle cycles while waiting for fetches from a higher level of the cache hierarchy. Going from 10% to 50% static energy contribution, sequential exploration with ratio projection revealed an additional 34% energy savings and the ACE-AWT heuristic showed an additional 40% energy savings.

The additional energy savings due to increased static power consumption can also soften the poor performance of inadequate tuning heuristics. Table 1 shows that for 50% static energy consumption, sequential exploration with ratio projection actually shows an average *energy savings* of 18% as opposed to the 24% increase in energy observed with the 10% static energy consumption. Whereas a tuning heuristic with an average energy savings of 24% is hardly a good heuristic compared to the ACE-AWT heuristic, this trend does suggest that tuning methodologies deemed as unsuccessful with today’s technology may seem more attractive as new technologies are revealed.

## 6. Tuning Environments

The ACE-AWT heuristic is primarily intended for use as a runtime optimization method for either desktop environments or embedded systems. However, the ACE-AWT heuristic is quite flexible and is easily applicable to all tuning environments such as a simulation-based configuration exploration or a hardware prototyping platform, as described in this section

The ACE-AWT heuristic is highly suitable for a dynamic runtime tuning environment for desktop environments or embedded systems. Zhang et al. [22] shows that level one cache tuning is feasible during runtime and the level one tuning in our work is based on Zhang’s tuning heuristic. Zhang shows that the actual tuning hardware adds very little area overhead. Zhang also explores the cache parameters such that cache flushing is minimized. However, for the cache flushing that does happen, we observe that flushing is very infrequent compared to the long run time needed to determine stabilized hit and miss rates for each cache configuration. Our level two configurable cache is based on the Motorola M\*CORE processor which was not allowed to have any overhead [15].

Because the ACE-AWT heuristic is a feasible dynamic runtime tuning heuristic, the tuning heuristic becomes more flexible to operating environments. The ACE-AWT heuristic can

be used to determine one low energy cache configuration to use throughout the entire run of an application by tuning once during startup. However, phase changes in applications suggest that different cache configurations are more appropriate for different execution phases of an application [12][18]. To better accommodate a single application environment with multiple phase changes, the tuning hardware would monitor the miss rates. When the miss rate exceeds a given threshold, the tuning hardware would reconfigure the cache for the new execution phase. To reduce tuning time, the heuristic cache configuration is saved and restored when the application reaches that execution phase again instead of rerunning the entire heuristic. Additionally, the ACE-AWT heuristic is suitable for a multi-application environment with an operating system. The tuning hardware would run each time an application swap occurs and, as with the application phase tuning, cache configurations are saved and restored to eliminate retuning when returning to a previously executed application. The minimization of the overhead incurred by runtime phase-based cache tuning and the implementation details are the focus of our future work.

In a hardware prototyping environment, two prototyping options exist - a full hardware prototyping environment and a platform assisted hardware prototyping environment. The full hardware prototyping environment consists of all tuning hardware implemented in hardware on the prototyping board. The tuning hardware would apply the ACE-AWT heuristic by running each cache configuration and measuring the hit and miss rates. Designer provided energy annotations guide the cache tuner to determine the next cache configuration to try. After completion of the heuristic, the best cache configuration can be reported to the designer. A platform assisted hardware prototyping environment couples a tunable platform with a PC to drive the tuning heuristic. The PC configures the platform for the configuration to try and then reads the hit and miss rates after a sufficiently long run of the application. The PC uses the cache hit and miss rates to drive the ACE-AWT heuristic and configure the platform for the next configuration to try.

In a simulation-based approach, application of the ACE-AWT heuristic is similar to the experimental environment set up for the results presented in this paper. Energy consumption estimates of cache and memory accesses are used to annotate the exploration heuristic. An exploration script is used in conjunction with a cache simulator to drive the heuristic. In addition to using a simulation approach for embedded systems, the simulation approach could also be used for profiling desktop computing environments.

Furthermore, the ACE-AWT heuristic is applicable in environments with other tunable parameters such as bus configuration and hardware/software partitioning by specifying a scheduling order for the configuration of the tunable parameters.

## 7. Conclusions and Future Work

We have presented an efficient method for cache hierarchy tuning for a highly configurable cache with a very large design space. The heuristic is designed to efficiently and accurately tune the level one and level two caches in a system during runtime but is also applicable to a hardware prototyping environment and a desktop simulation cache exploration environment. Our heuristic determines a cache configuration that consumes on average 62% less energy than a base cache configuration while exploring only 0.2% of the design space. Additionally, our cache tuning results in

an average speedup of 35% due to line size configuration. We also show the fidelity of our tuning heuristic across future technologies with increasing static power consumption. Future work includes recompilation of the application to the best cache configuration for further energy and performance benefits. Additionally, the heuristic can be expanded to produce a Pareto optimal set of cache configurations trading off energy and performance. We also plan to examine desktop and mainframe applications on appropriate cache configurations for different application execution phases.

## 8. References

- [1] Albonese, D.H. Selective cache ways: on demand cache resource allocation. *Journal of Instruction Level Parallelism*, May 2002.
- [2] Altera, Nios Embedded Processor System Development, [http://www.altera.com/corporate/news\\_room/releases/products/nr-nios\\_delivers\\_goods.html](http://www.altera.com/corporate/news_room/releases/products/nr-nios_delivers_goods.html)
- [3] Arc International, [www.arccores.com](http://www.arccores.com).
- [4] ARM, [www.arm.com](http://www.arm.com).
- [5] Balasubramonian, R., Albonese, D., Buyuktosunoglu, A., Dwarkadas, S. Memory heirarchy reconfiguration for energy and performance in general-purpose processor architecture. 33rd International Symposium on Microarchitecture, December 2000.
- [6] Burger, D., Austin, T., Bennet, S. Evaluating future microprocessors: the simplescalar toolset. University of Wisconsin-Madison. Computer Science Department Tech. Report CS-TR-1308, July 2000.
- [7] EEMBC, the Embedded Microprocessor Benchmark Consortium, [www.eembc.org](http://www.eembc.org).
- [8] Givargis, T., Vahid, F. Platune: a tuning framework for system-on-a-chip platforms. *IEEE Transactions on Computer Aided Design*, November 2002.
- [9] Ghosh, A., Givargis, T. Cache optimization for embedded processor cores: an analytical approach. *International Conference on Computer Aided Design*, November 2003.
- [10] 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.
- [11] 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.
- [12] Merten, M.C., Trick, A.R., George, C.N., Gyllenhaal, J., Hwu, W.W. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization, In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, 1999.
- [13] MIPS Technologies, [www.mips.com](http://www.mips.com).
- [14] Palesi, M., Givargis, T. Multi-objective design space exploration using genetic algorithms. *International Workshop on Hardware/Software Codesign*, May 2002.
- [15] Personal communication with M\*CORE designers
- [16] Reinman, G., Jouppi, N.P. Cacti2.0: an integrated cache timing and power model. *COMPAQ Western Research Lab*, 1999.
- [17] Segars, S. Low power design techniques for microprocessors, *International Solid State Circuit Conference*, February 2001.
- [18] 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
- [19] Tensilica, Xtensa Processor Generator, <http://www.tensilica.com/>.
- [20] Veidenbaum, A., Tang, W., Gupta, R., Nicolau, A., Ji, X. Adapting cache line size to application behavior. *International Conference on Supercomputing*, June 1999.
- [21] Zhang, C., Vahid, F., Najjar, W. A highly-configurable cache architecture for embedded systems. *30th Annual International Symposium on Computer Architecture*, June 2003.
- [22] Zhang, C., Vahid, F. A self-tuning cache architecture for embedded systems. *Design, Automation and Test Conference in Europe (DATE)*, 2004