Fast Configurable-Cache Tuning with a Unified Second-Level Cache[1]

Ann Gordon-Ross, *Member IEEE,* Frank Vahid, *Senior Member IEEE,* and Nikil Dutt, *Fellow IEEE*

***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 cache, 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 61% energy savings and 9% performance improvements over a non-configurable cache, greatly outperforming an extension of a previous method.**

***Index Terms*—Configurable cache hierarchy, cache exploration, cache optimization, low power, low energy, architecture tuning, and embedded systems.**

I. Introduction

The memory hierarchy of a microprocessor can consume as much as 50% of the system power in a microprocessor [15][20]. 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 desktops and mainframes where chips are requiring costly cooling methods.

Applications require highly diverse cache configurations for optimal energy consumption in the memory hierarchy [26]. Even different phases of the same application may benefit from different cache configurations in each phase [16][21]. 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 designate a specific cache configuration [2][3][4][17][22]. Additionally, processors with configurable caches are available that can have their caches configured during system reset or even during runtime [1][15][26]. Such configurable caches have been shown to have very little size or performance overhead compared to non-configurable caches [15][24].

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% [24]. 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% [12]. The increased savings suggest that increasing the 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*, rather than separate (having one cache for instructions and another for data). A multi-way unified cache

Manuscript received May 10, 2006. This work was supported in part by the National Science Foundation (CCR-0203829, CCR-9876006) and by the Semiconductor Research Corporation (2005-HJ-1331).

Ann Gordon-Ross is with the University of Florida, Gainesville, Fl. 32608 USA. She is a member of the NSF Center for High-Performance Reconfigurable Computing (CHREC). (phone: 352-392-5356; e-mail: ann@ece.ufl.edu; home page: http://www.ann.ece.ufl.edu/).

Frank Vahid is with the University of California, Riverside, Riverside, Ca 92521 USA. He is also with the Center for Embedded Computer Systems at UC Irvine. (e-mail: vahid@cs.ucr.edu; home page: http://www.cs.ucr.edu/~vahid).

Nikil Dutt is with the University of California, Irvine, Irvine, Ca 92697. (e-mail: dutt@cecs.uci.edu; home page: http://www.ics.uci.edu/~dutt).

[1] This paper is an extension to a paper that appeared in the International Symposium on Low Power Electronics and Design, 2005 [13].

enables tradeoffs between the number of instruction ways and the number of data ways, with those tradeoffs known as way management [15]. 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 1 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 the 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 in part 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 approximately 18,000 possible cache configurations. Our heuristic achieves an average energy savings of 61%, 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 of cache configuration heuristics. We design our heuristic to be lightweight enough to be implemented in an on-chip dynamic tuning approach without imposing excess overhead and flexible enough to be used in a variety of different tuning environments.

II. Related Work

Commercial systems with tunable caches (e.g., [4][15]) 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

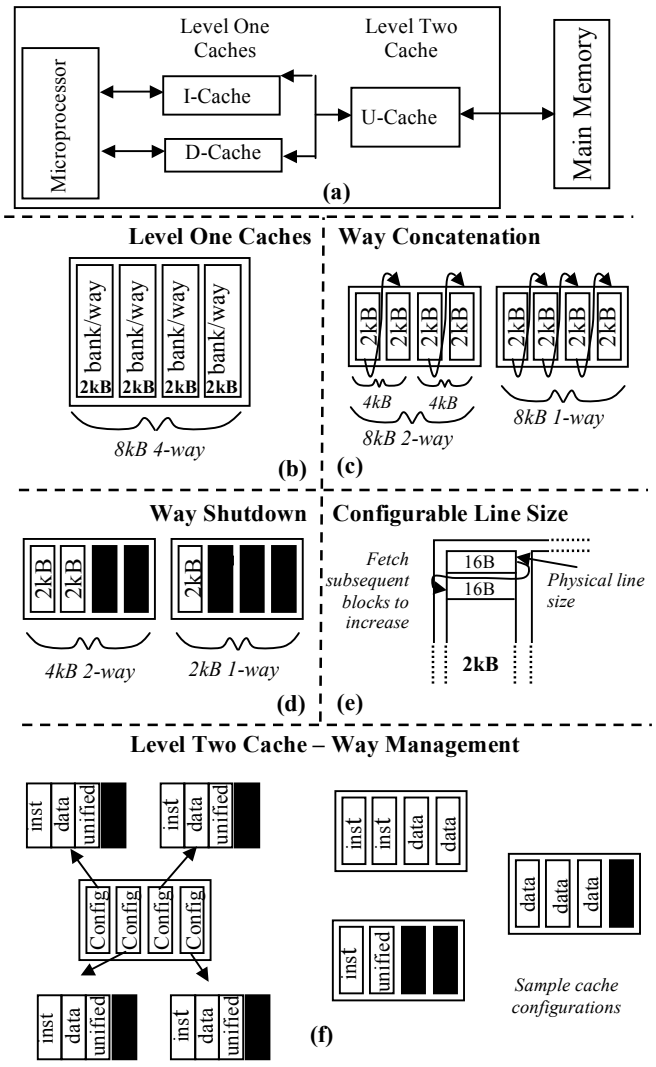

Fig. 1. Configurable Cache Architecture: (a) system architecture and cache hierarchy used, (b) base cache bank layout for the level one caches, (c) way concatenation offered in level one caches, (d) way shutdown offered in level one caches, (e) configurable line size offered in all caches, and (f) configurability available for the level two cache utilizing way management.

efforts focus on single level cache tuning. Platune [10] 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, interdependent parameters 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. [18] 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. [24] 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. [11] presents a heuristic that, through an analytical model, directly determines the cache configuration based on the designers performance constraints. Ge et al. [9] present an algorithm for partitioning a fixed about of reconfigurable on-chip storage resources between the instruction cache and a scratch pad memory for energy savings averaging 30%.

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 level one and level two cache reconfiguration as well as redistributing the cache size between the level two and level three caches while maintaining a conventional level one cache. In previous work [12], 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. Dhodapkar et al. [7] present a method to dynamically monitor working set characteristics and infer program resource requirements for a multi-level cache hierarchy. On-chip profiling hardware calculates a certain miss penalty threshold for each configurable unit and a greedy tuning heuristic explores the design space for each unit until a configuration that meets the miss penalty threshold is determined.

III. Configurable Cache Architecture

Fig. 1 (a) depicts our target system architecture. On-chip components consist of a microprocessor connected to separate level one instruction and data caches, each of which connects to a unified level two cache. The level two cache connects to an off-chip main memory.

A. Level One Caches

Fig. 1 (b) illustrates the level one configurable cache architecture based on the tunable cache described by Zhang et al. [25][26]. The base cache structure is an 8 KB cache consisting of four 2 KB banks where each bank acts as a separate way – thus the base cache is an 8 KB, 4-way set associative cache. Zhang provides hardware layout verification for the configurable cache and shows that the configuration circuitry does not increase the access time of the cache. The tunable parameters consist of cache size, line size, and associativity.

Fig. 1 (c) depicts way concatenation. Special way configuration registers allow for banks to be logically concatenated thus enabling associativity configurability. Fig. 1 (c) shows a 2-way set associative and a direct-mapped (1-way set associative) cache using way concatenation.

Additionally, banks/ways may be shut down to enable configurable size, as depicted in Fig. 1 (d). Way shut down and way concatenation may be combined to offer other combinations of size and associativity. However, due to the bank layout of the cache, 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 simulation-based exploration, any cache configuration is possible.

Fig. 1 (e) depicts the configurable line size available in both the level one and level two caches. The configurable cache consists of a base physical line size of 16 bytes and is configurable to 32 and 64 bytes by fetching subsequent blocks in memory.

Fig. 2 shows the impact-ordered heuristic developed by Zhang to efficiently explore the highly configurable level one cache. Through experimental results, Zhang determined that the cache parameters should be explored in order of their impact on total energy, with the highest impact parameter explored first followed by the second highest, and so on. This impact ordering of parameters explores total size, followed by line size, and then followed by associativity. For each parameter, the heuristic explores the parameters values from smallest to largest to minimize the number of cache flushes in a runtime tuning environment. For each cache parameter, values are successively explored until there is no reduction in energy revealed, and thus the previous value explored resulted in the lowest energy consumption for that cache parameter. That cache parameter value becomes fixed at the value that revealed the lowest energy consumption.

In later work, we extended Zhang's heuristic to explore two levels of cache where the second level of cache consisted of separate instruction and data caches [12]. We observed that whereas exploring each successive parameter value as long as a decrease in energy is observed was sufficient for a single level of cache, this limited potential savings in a two level cache due to dependencies between the level one and level two caches. We improved upon the heuristic in [12] by fully exploring all parameter values. This simply required removing the else portion of the if statements in each for

```
best_size = SMALLEST_SIZE
best_assoc = SMALLEST_ASSOC
best_linesize = SMALLEST_LINESIZE
current_smallest_energy = ∞

foreach available_size // searching from smallest to largest
    energy = simulate_cache ( available_size, best_assoc, best_linesize)
    if ( energy < current_smallest_energy )
        current_smallest_energy = energy
        best_size  = available_size
    else
        break

foreach available_linesize // searching from smallest to largest
    energy = simulate_cache ( best_size, best_assoc, available_linesize)
    if ( energy < current_smallest_energy )
        current_smallest_energy = energy
        best_linesize  = available_linesize
    else
        break

foreach possible_assoc // searching from smallest to largest
    energy = simulate_cache ( best_size, possible_assoc, best_linesize)
    if ( energy < current_smallest_energy )
        current_smallest_energy = energy
        best_assoc  = possible_assoc
    else
        break
```

Fig. 2. Impact-ordered heuristic for exploring level one cache parameters. The same heuristic is applied to both the instruction and data caches

loop.

B. Level Two Cache

The second level cache is a configurable unified cache quite different than the first level cache, illustrated in Fig. 1 (f). For the second level, we utilize way management implemented in Motorola's M*CORE processor [15]. In a way management cache, each way is a configurable way and may be designated as a unified way, an instruction-only way, a data-only way, or the way can be shut down entirely.

C. Cache Parameter Values and Configuration Space

For the cache parameters values, 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. Additionally, our chosen values result in a wide variety of resulting best cache configurations across the benchmarks studied, showing that the parameter values utilized covers the needs of many different applications. 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 approximately 18,000 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, 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 40,000. However, restrictions reduce the number of configurations to approximately 18,000. For example, the second level line size must be greater than or equal to the largest level one line size.

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 13 selected 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 caches and a 64 Kbyte fully unified cache with a 64 byte line size for the level two cache – a reasonably common configuration.

IV. Tuning Heuristics

For our configurable cache hierarchy, the full configuration space consists of nearly 18,000 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.

A. Sequential Exploration with Ratio Projection - SERP

A simple tuning heuristic for two-level caches ignores all tuning dependency between the level one instruction and data caches and the level one and level two 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. Fig. 3 summarizes are our first heuristic sequential exploration with ratio projection (SERP).

For level one exploration, SERP utilizes the impact ordering of parameters and exploration ordering of parameter values including full parameter exploration as described in section III.A.

For the level two cache, SERP must also consider that the cache offers way management. Thus, not only must the heuristic determine the total size, line size, and associativity, but the heuristic must also determine how many ways will be for data, how many for instructions, how many for both instructions and data, and how many will be shut down. For

```
best_L1_I_size = SMALLEST_L1_SIZE
best_L1_D_size = SMALLEST_L1_SIZE
best_L1_I_assoc = SMALLEST_L1_ASSOC
best_L1_D_assoc = SMALLEST_L1_ASSOC
best_L1_I_linesize = SMALLEST_L1_LINESIZE
best_L1_D_linesize = SMALLEST_L1_LINESIZE
best_L2_linesize = SMALLEST_L2_LINESIZE
L2_way_configuration = UEEE   // one unified way and 3 ways shutdown
current_smallest_energy = ∞

// explore_L1_cache function calls the impact-ordered heuristic outlined
// in Fig. 2
explore_L1_cache ( ICACHE, &best_L1_I_size, &best_L1_I_assoc,
                   &best_L1_I_linesize, &best_L2_linesize,
                   &current_smallest_energy, … )
explore_L1_cache ( DCACHE, &best_L1_D_size, &best_L1_D_assoc,
                   &best_L1_D_linesize, &best_L2_linesize,
                   &current_smallest_energy, … )

// searching from smallest to largest – ensure L2 line size is greater than
// L1 line sizes
foreach available_L2_linesize
   energy = simulate_cache ( available_L2_linesize, …  )
   if ( energy < current_smallest_energy )
      current_smallest_energy = energy
      best_L2_linesize  = available_L2_linesize

// explore L2 cache using ratio projection outlined in Fig. 4
ratio_projection ( &L2_way_configuration, &current_smallest_energy )
```

Fig. 3. Sequential exploration with ratio projection (SERP) heuristic. (…) in function calls passes all other necessary cache parameter values

unified level two cache exploration, we initially developed the ratio projection portion of SERP.

The ***ratio projection*** method, illustrated in Fig. 4, projects the number of necessary instruction and data ways needed for the best cache configuration. During level two cache exploration, SERP executes 6 different configurations to gather information on the instruction and data caching requirements of the level two cache. To explore an application's instruction caching needs, SERP sets the level two cache to have one data way and adds instructions ways one at a time (Fig. 4 (a)). The lowest energy configuration suggests the ideal number of instruction ways needed in the level two cache. Similarly, SERP determines the ideal number of data ways. Ratio projection then combines the ideal number of instruction and data ways to determine the ideal level two way designations. Simply adding the number of ways could exceed the available number of ways in the level two cache. In the situation where the ideal number of ways exceeds the number of ways in the level two cache, ratio projection must carefully combine the instruction and data ways to keep the *ratio* of instruction to data ways as close to the ideal as possible while meeting the constraints of the level two cache. Keeping the ratio in mind will allow for the more important way type (the way designation (instruction or data) with the larger number of ideal ways) to be allocated more ways in the final level two configuration.

There are two situations that may occur during ratio projection. The first situation occurs when both the instructions and data are equally important in the level two cache – the number of ideal ways is equal. In this case, we use way reduction and simply remove 1 data and 1 instruction way at a time until the combined number of ways is less than the total number of ways available in the level two cache. For example, ratio projection might determine the ideal number of instruction and data ways to be 3 and 3, respectively. Given only four available ways, ratio projection would allocate 2 instruction and 2 data ways, thus maintaining the same ratio of instruction to data ways.

The second situation occurs when one way designation (instruction or data) is more important than the other – the ideal number of ways is different. In this case, we cannot simply use way reduction to remove 1 data and 1 instruction way until the combined number of ways is less than the total number of available level two ways. This reduction may lead to undesignated ways. For example, if the ideal number of instruction ways is 2 and the ideal number of data ways is 3, removing 1 way of each type would result in the level two cache having 1 instruction way, 1 data way, and 1 way shut down. Additionally, in level two cache configurations offering more than 4 total ways, this method may cause either instructions or data not to have any level two designations. This situation may occur if there were 8 available level two ways and the ideal number of instruction and data ways are 1 and 8 respectively. We could alternatively only remove 1 data way or 1 instruction way, but this would not maintain the ideal ratio of instruction to data ways and choosing which way to remove becomes arbitrary. To resolve this situation, we use way unification (illustrated in Fig. 4 (c)) to determine our final level two way designations. Instead of completely removing 1 instruction and 1 data way, we *unify* an instruction way with a data way, reducing the total number of required ways by 1. We continue to make this reduction with unification until the combined number of ways is less than the total number of ways available in the level two cache.

Through extensive experimentation, we observed that the SERP produced substandard results for many benchmarks. Although the heuristic resulted in a 37% decrease in energy consumption over the base cache configuration, for a few examples the energy consumption *increased*. Given the vastly increased configuration space over previous methods, we had expected to see significant additional energy savings, when in fact, SERP revealed *less* energy savings than previous methods. Previous methods with separate level one and level two caches showed 53% energy savings on average [12]. Clearly, a simple adaptation of current methods does not sufficiently explore tuning dependencies.

B. 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

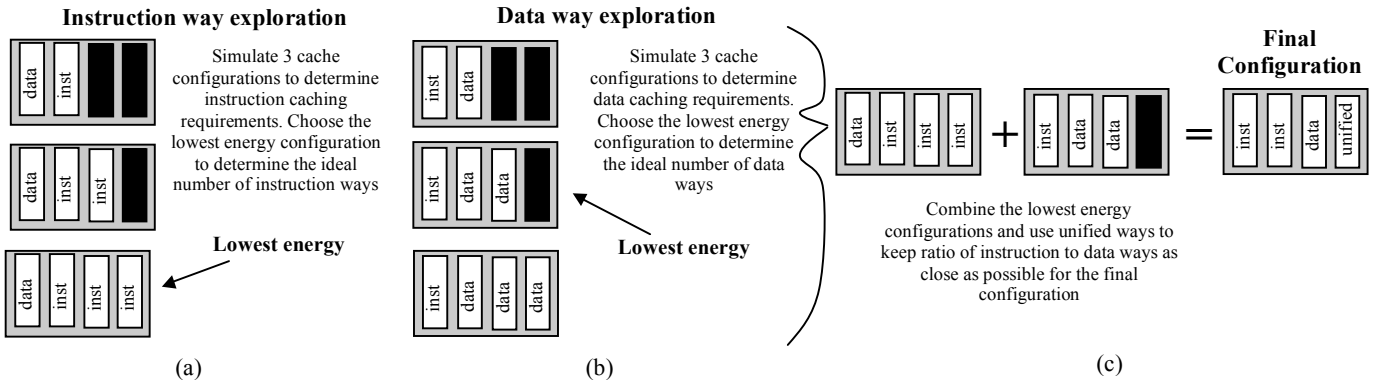

Fig. 4. Ratio projection for level two cache way exploration showing a sample reduction with way unification.

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 [12], we similarly concluded the importance of tuning both cache levels 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 that parameter for the level two cache, before proceeding to explore the next parameter. However, that interlaced method only addressed the dependency between separate level one and level two caches, and not the dependency between the level one instruction and data caches. We further explore the level one dependencies in section V.C. 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 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 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) and is illustrated in Fig. 5. 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 Fig. 6 (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 candidate 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 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 level two line size while increasing 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, Fig. 6 (b) illustrates the final tuning step applied to fine tune the cache configuration.

```
best_L1_I_size = SMALLEST_L1_SIZE
best_L1_D_size = SMALLEST_L1_SIZE
best_L1_I_assoc = SMALLEST_L1_ASSOC
best_L1_D_assoc = SMALLEST_L1_ASSOC
best_L1_I_linesize = SMALLEST_L1_LINESIZE
best_L1_D_linesize = SMALLEST_L1_LINESIZE
best_L2_linesize = SMALLEST_L2_LINESIZE
L2_way_configuration = UEEE   // one unified way and 3 ways shutdown
current_smallest_energy = ∞

// explore L1 I size
foreach available_size // searching from smallest to largest
   energy = simulate_cache ( available_size, … )
   if ( energy < current_smallest_energy )
      current_smallest_energy = energy
      best_L1_I_size  = available_size

// explore L1 D size
foreach available_size // searching from smallest to largest
   energy = simulate_cache ( available_size, … )
   if ( energy < current_smallest_energy )
      current_smallest_energy = energy
      best_L1_D_size  = available_size

// ACE-AWT first phase - explore L2 size outlined in Fig. 6(a)
ACE-AWT_first_phase ( &L2_way_configuration,
                      &current_smallest_energy, … )

// explore L1 I line size
foreach available_linesize // searching from smallest to largest
   energy = simulate_cache ( available_linesize, … )
   if ( energy < current_smallest_energy )
      current_smallest_energy = energy
      best_L1_I_linesize  = available_linesize

// explore L1 D line size
foreach available_linesize // searching from smallest to largest
   energy = simulate_cache ( available_linesize, … )
   if ( energy < current_smallest_energy )
      current_smallest_energy = energy
      best_L1_D_linesize  = available_linesize

// explore L2 line size – ensure L2 line size is greater than L1 line sizes
foreach available_linesize // searching from smallest to largest
   energy = simulate_cache ( available_linesize, … )
   if ( energy < current_smallest_energy )
      current_smallest_energy = energy
      best_L2_linesize  = available_linesize

// explore L1 I assoc
foreach available_assoc // searching from smallest to largest
   energy = simulate_cache ( available_assoc, … )
   if ( energy < current_smallest_energy )
      current_smallest_energy = energy
      best_L1_I_assoc  = available_assoc

// explore L1 D assoc
foreach available_assoc // searching from smallest to largest
   energy = simulate_cache ( available_assoc, … )
   if ( energy < current_smallest_energy )
      current_smallest_energy = energy
      best_L1_D_assoc  = available_assoc

//ACE-AWT fine tuning phase – explore L2 size outlined in Fig. 6(b)
ACE-AWT_fine_tuning_phase ( &L2_way_configuration,
                            &current_smallest_energy, … )
```

Fig. 5. Alternating cache exploration with additive way tuning (ACE-AWT) heuristic. (…) in function calls passes all other necessary cache parameter values

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, the heuristic ignores the reduction step. The lowest energy cache configuration is chosen if it consumes less energy than 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.

Since the fine-tuning phase iteratively adds and removes ways, this results in identical cache configurations being explored during different iterations of the fine-tuning phase. To eliminate redundant exploration of previously explored cache configurations, we record each cache configuration explored. However, in the worst case, the ACE-AWT heuristic may explore 88 cache configurations.

V. RESULTS

A. Experimental Setup

We applied each heuristic to 16 benchmarks from the EEMBC benchmark suite [8], 12 benchmarks from the Powerstone benchmark suite [15], and 6 benchmarks from the MediaBench benchmark suite [14]. 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 determine energy consumption for a cache configuration for both static and dynamic energy using the following model:

total_energy = static_energy + dynamic_energy
dynamic_energy = cache_hits * hit_energy + cache_misses * miss_energy
miss_energy = offchip_access_energy +miss_cycles * CPU_stall_energy + cache_fill_energy
miss_cycles = cache_misses * miss_latency + (cache_misses * (linesize/16) * memory_bandwidth)
static_energy = total_cycles * static_energy_per_cycle
static_energy_per_cycle = energy_per_Kbyte * cache_size_in_Kbytes
energy_per_Kbyte = ((dynamic_energy_of_base_cache * 10%) / base_cache_size_in_Kbytes)

We used Cacti [19] to determine the dynamic energy consumed by each cache fetch for each cache configuration using 0.18-micron technology. We used SimpleScalar [6] to measure cache hits and cache misses for each cache configuration. Miss energy determination is quite difficult because it depends on the off-chip access energy and the CPU stall energy which are highly dependent on the actual system configuration used. We could have chosen a particular system configuration and obtained hard values for the *CPU_stall_energy* however, our results would only apply to one particular system configuration. Instead, we examined the stall energy for several microprocessors and estimate the *CPU_stall_energy* to be 20% of the active energy of the microprocessor for this study. We obtain the *offchip_access_energy* from a standard low-power Samsung memory. To obtain miss cycles, the miss latency and bandwidth of the system is required. 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 twenty times slower than a level two fetch. We assume memory throughput is 10% of the latency, meaning blocks fetched after the first block take 10% of the latency of the first block fetched. In previous work [12], we showed that cache tuning heuristics remain valid across different configurations of miss latency and bandwidth. We determine the static energy per Kbyte as 10% of the dynamic energy of the base cache divided by the base cache size in Kbytes. In section 0, we explore the impact of increasing static energy consumption on cache configuration heuristics.

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.

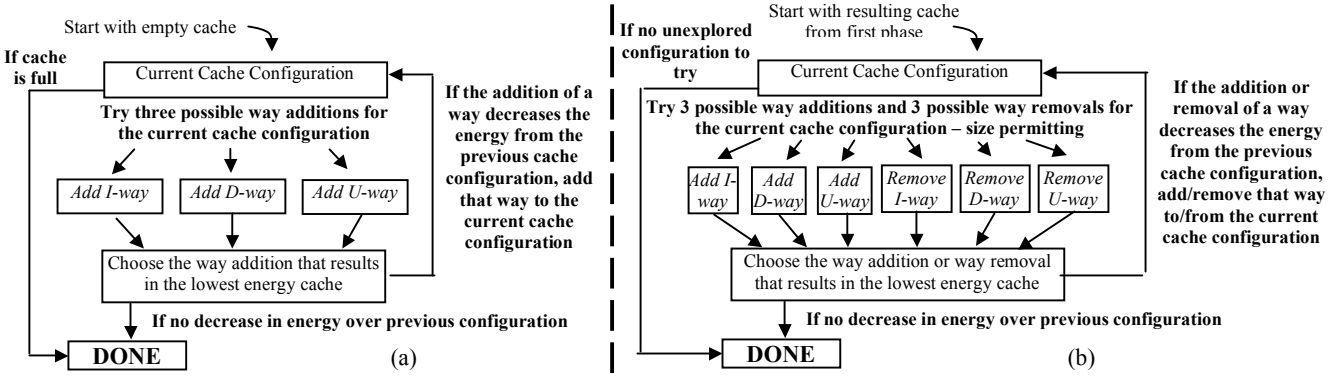

Fig. 6. Additive way tuning for level two cache way exploration for the (a) first phase and (b) the fine tuning phase.

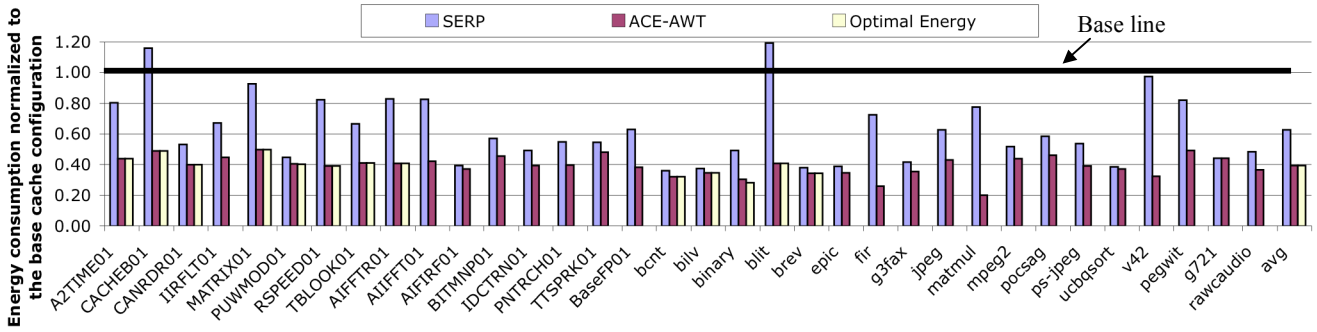

Fig. 7. Energy consumption normalized to the base cache configuration for both cache exploration heuristics and the optimal cache configuration.

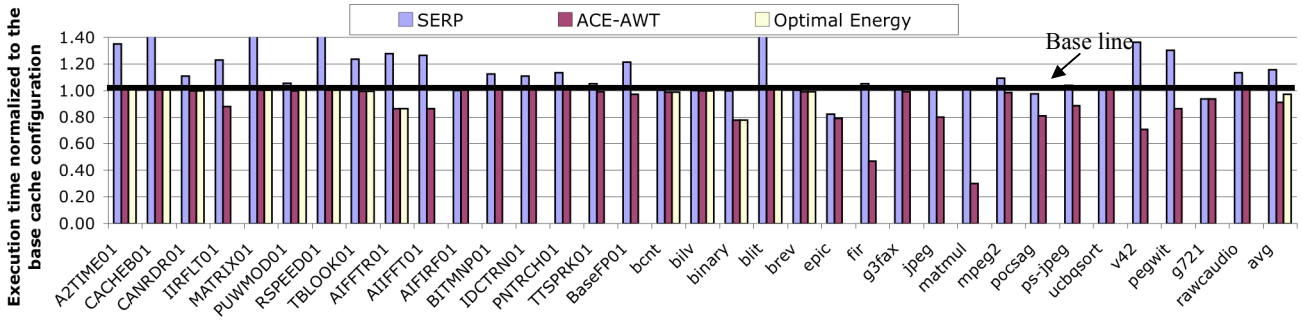

Fig. 8. Execution time normalized to the base cache configuration for both exploration heuristics and the optimal cache configuration

B. *Energy Consumption and Performance*

Fig. 7 shows the energy consumption for all benchmarks for both tuning heuristics and the optimal cache configuration for 13 benchmarks. Energy consumption for each configuration is normalized to the energy consumption of the base cache for that benchmark. Fig. 7 shows that while SERP achieved average energy savings of 37%, the energy consumption actually *increased* for two benchmarks. The ACE-AWT heuristic improves greatly over SERP showing energy savings of 61% averaged over all benchmarks. For the 13 benchmarks where the optimal cache configuration is known, ACE-AWT either finds the optimal cache configuration or determines a cache configuration that is very near the optimal. 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.

As well as showing good energy savings across all benchmarks, we examine the performance impact of the ACE-AWT heuristic. Fig. 8 shows the execution time of each benchmark for the ACE-AWT heuristic normalized to the execution time for the base cache configuration. On average, the ACE-AWT heuristic, while tuning solely for energy, achieves a 9% performance improvement. Each benchmark either shows an improvement in performance or a very minute decrease in performance. We found that this improvement is due to tuning the line size to the locality needs of the application [12].

To ensure that the energy savings obtained with ACE-AWT are not inflated due to choosing a base cache configuration that consumes a large amount of energy, we examined the energy savings obtained by ACE-AWT compared to the highest energy consuming cache. Fig. 9 shows the energy consumption of the ACE-AWT configuration normalized to the worst energy consuming cache for each benchmark that we exhaustively explored. ACE-AWT achieves average energy savings of 94% when compared to the worst energy configuration. Thus, an average energy savings of 61% compared to a base cache configuration does not unnecessarily inflate our results.

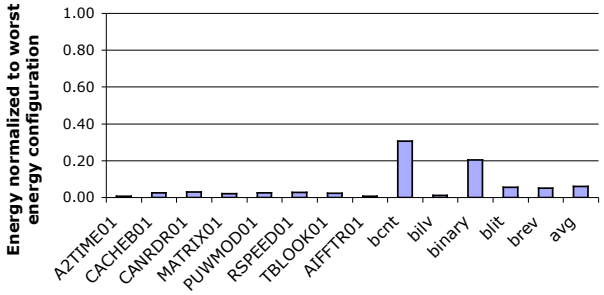

Fig. 9. Energy consumption of the ACE-AWT configuration normalized to the worst energy consuming cache for each benchmark.

Additionally, Fig. 10 shows the execution time of the ACE-AWT configuration normalized to the execution time of the best performing cache for each benchmark. Excluding the *CACHEB01* benchmark, the ACE-AWT configuration increases execution time by at most only 5% compared to the best performing cache.

C. *Level One Instruction and Data Cache Dependencies*

During interlacing, the ACE-AWT heuristic explores the level one instruction parameter followed by the level one data parameter. Due to the dependencies between the level

one caches via the level two cache, this ordering of instruction exploration before data exploration can impact the results.

Fig. 11 shows the impact that this ordering has on the potential energy savings and performance benefits revealed by the ACE-AWT heuristic. Fig. 11 (a) shows the energy consumption normalized to the base cache configuration for the ACE-AWT heuristic both exploring the instruction parameter before the data parameter and the data parameter before the instruction parameter. On average, both methods achieve 61% energy savings. In 14 benchmarks, exploring the instruction parameter before the data parameter either produced identical or better savings that exploring the data parameter before the instruction parameter. In 20 benchmarks, exploring the data parameter before the instruction parameter produced better savings. In a few benchmarks, the difference in savings is quite large, however, no one method consistently outperformed the other by a large amount.

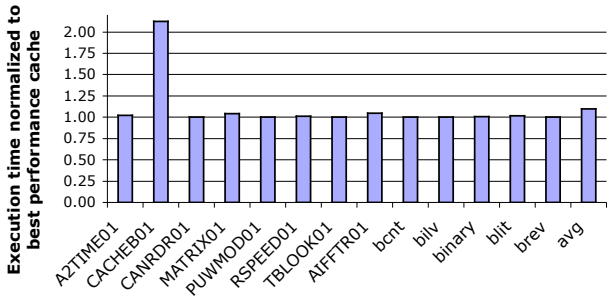

Fig. 10. Execution time of the ACE-AWT configuration normalized to the execution time of the best performance configuration for each benchmark.

Fig. 11 (b) shows the execution time normalized to the base cache configuration for the ACE-AWT heuristic for both orderings of exploration showing large differences between the two methods. On average, exploring the instruction parameter first yields a 9% reduction in execution time, while exploring the data parameter first *increases* the average execution time by 3%. Studying the benchmarks more closely reveals that only 9 benchmarks perform better when the data parameter is explored first and in the cases where one method significantly outperforms the other, exploring the instruction parameter first consistently performed better.

Taking into consideration both energy savings and performance benefits, we conclude that on average, the instruction parameter should be explored before the data parameter. However, since the ACE-AWT heuristic explores a very small number of configurations and if energy savings is paramount to performance, it would be feasible to apply the heuristic twice, once exploring the instruction parameter first and once exploring the data parameter first, and then choosing the lowest energy of the two configurations.

D. Cache Configurations

TABLE I shows the cache configurations chosen by both exploration heuristics and the optimal energy cache. In the cases where the optimal energy cache is known. We point out that the ACE-AWT does not necessarily find the optimal energy cache even though the energy savings may be appear identical in Fig. 7. In a design space consisting of 18,000 configurations, there is one configuration with the lowest energy consumption and there are a number of configurations

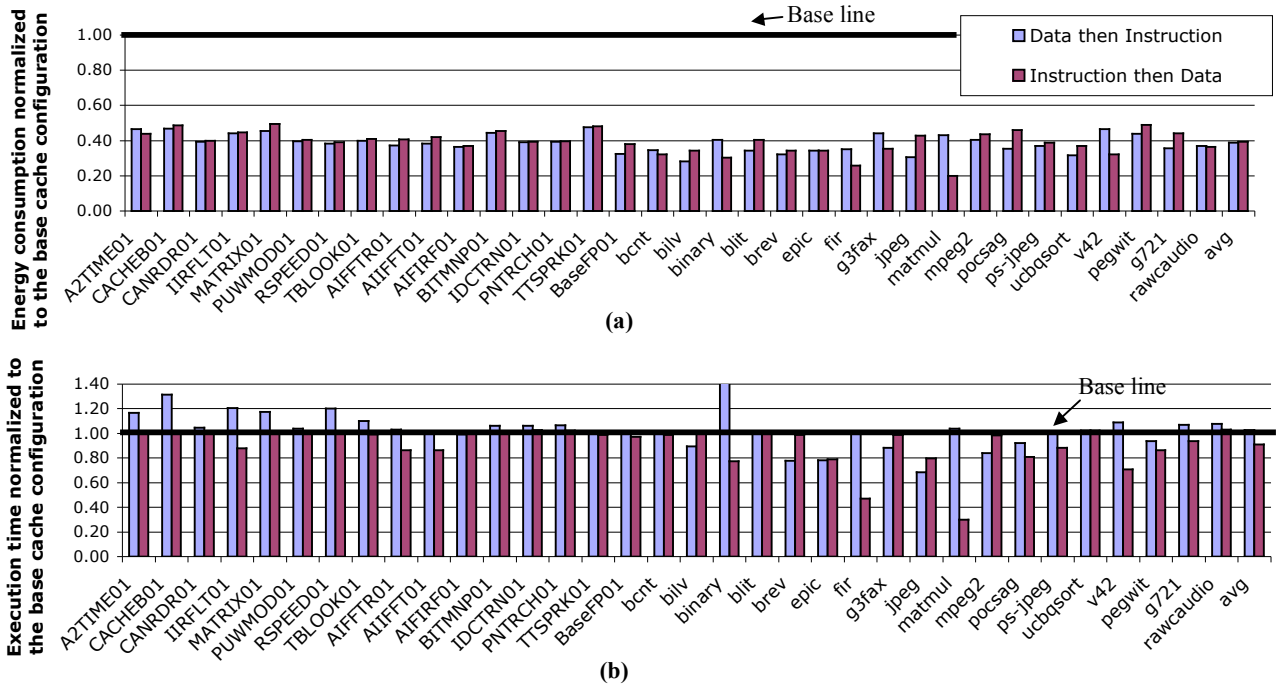

Fig. 11. Energy consumption (a) and execution time (b) normalized to the base cache configuration for the ACE-AWT heuristic comparing interlacing methods that explore the data parameter before the instruction parameter and the instruction parameter before the data parameter

TABLE I

CACHE CONFIGURATIONS CHOSEN BY BOTH EXPLORATION HEURISTICS AND THE OPTIMAL ENERGY CACHE. THE LEVEL ONE INSTRUCTION (IL1), LEVEL ONE DATA (DL1), AND LEVEL TWO UNIFIED (UL2) CACHE CONFIGURATIONS ARE LISTED AS THE TOTAL SIZE IN KBYTES (2, 4, OR 8 K) FOLLOWED BY THE ASSOCIATIVITY (1, 2, OR 4 WAY (W)) FOLLOWED BY THE LINE SIZE IN BYTES. THE LEVEL TWO CACHE WAY DESIGNATIONS ARE SPECIFIED AS INSTRUCTION (I), DATA (D), UNIFIED (U), OR SHUT-DOWN (E – EMPTY).

Benchmark	SERP	ACE-AWT	Optimal Energy
A2TIME01	il1_8k1w64_dl1_4k1w16_ul2_32k4w64_DIUU	il1_8k1w32_dl1_2k1w16_ul2_32k4w32_DDEI	il1_8k1w64_dl1_4k1w16_ul2_24k3w64_DDEU
CACHEB01	il1_8k1w64_dl1_8k1w16_ul2_32k4w64_DIUU	il1_8k1w16_dl1_8k1w16_ul2_32k4w16_DDEU	il1_8k1w16_dl1_8k1w16_ul2_24k3w16_DDEU
CANRDR01	il1_4k1w64_dl1_8k1w16_ul2_32k4w64_DIUU	il1_4k1w32_dl1_8k1w16_ul2_32k4w32_DDEU	il1_4k1w32_dl1_8k1w16_ul2_24k3w32_DDEU
IIRFLT01	il1_2k1w64_dl1_4k1w32_ul2_32k4w64_DDDI	il1_8k1w16_dl1_4k1w16_ul2_32k4w16_EEEU	
MATRIX01	il1_2k1w32_dl1_8k1w16_ul2_32k4w32_DIUU	il1_2k1w16_dl1_8k1w16_ul2_32k4w16_DEUU	il1_2k1w16_dl1_8k1w16_ul2_24k3w16_DEUU
PUWMOD01	il1_4k1w64_dl1_2k1w16_ul2_32k4w64_DIUU	il1_4k1w32_dl1_8k1w16_ul2_32k4w32_DDEU	il1_4k1w64_dl1_8k1w64_ul2_24k3w64_DDEU
RSPEED01	il1_4k1w64_dl1_4k1w16_ul2_32k4w64_DIUU	il1_4k1w32_dl1_4k1w16_ul2_32k4w32_DDEU	il1_4k1w32_dl1_4k1w16_ul2_24k3w32_DDEU
TBLOOK01	il1_4k1w64_dl1_8k1w16_ul2_32k4w64_DIUU	il1_4k1w32_dl1_8k1w16_ul2_32k4w32_DDEU	il1_4k1w32_dl1_8k1w16_ul2_24k3w32_DDEU
AIFFTR01	il1_2k1w32_dl1_8k1w16_ul2_32k4w32_DIUU	il1_2k1w16_dl1_8k1w16_ul2_32k4w16_DEUU	il1_2k1w16_dl1_8k1w16_ul2_24k3w16_DEUU
AIIFFT01	il1_2k1w32_dl1_8k1w16_ul2_32k4w32_DIUU	il1_2k1w16_dl1_8k1w16_ul2_32k4w16_DEIU	
AIFIRF01	il1_4k1w64_dl1_4k1w32_ul2_32k4w64_DIUU	il1_4k1w64_dl1_2k1w16_ul2_32k4w64_DDEI	
BITMNP01	il1_8k1w64_dl1_4k1w16_ul2_32k4w64_DIUU	il1_8k1w16_dl1_4k1w16_ul2_32k4w16_DEII	
IDCTRN01	il1_2k1w32_dl1_4k1w16_ul2_32k4w32_DIIU	il1_2k1w32_dl1_4k1w16_ul2_32k4w32_DEIU	
PNTRCH01	il1_2k1w32_dl1_8k1w16_ul2_32k4w32_DIUU	il1_2k1w32_dl1_4k1w16_ul2_32k4w32_DDEU	
TTSPRK01	il1_8k1w32_dl1_8k1w16_ul2_32k4w32_DIUU	il1_8k1w16_dl1_8k1w16_ul2_32k4w16_DDEU	
BaseFP01	il1_4k1w64_dl1_2k1w16_ul2_32k4w64_DIUU	il1_4k1w32_dl1_2k1w16_ul2_32k4w32_DDEU	
bcnt	il1_2k1w64_dl1_2k1w64_ul2_32k4w64_DIUU	il1_2k1w32_dl1_2k1w64_ul2_32k4w64_DDEU	il1_2k1w32_dl1_2k1w64_ul2_24k3w64_DDEU
bilv	il1_4k1w64_dl1_2k1w64_ul2_32k4w64_DIUU	il1_4k1w64_dl1_2k1w64_ul2_32k4w64_DDEU	il1_4k1w64_dl1_2k1w64_ul2_24k3w64_DDEU
binary	il1_2k1w32_dl1_2k1w64_ul2_32k4w64_DIUU	il1_2k1w32_dl1_8k1w32_ul2_32k4w32_DEIU	il1_2k1w16_dl1_2k1w16_ul2_24k3w16_DDEI
blit	il1_2k1w64_dl1_2k1w16_ul2_32k4w64_DIUU	il1_2k1w32_dl1_8k1w16_ul2_32k4w64_DDEU	il1_2k1w32_dl1_8k1w16_ul2_24k3w32_DDEU
brev	il1_4k1w64_dl1_2k1w64_ul2_32k4w64_DIUU	il1_4k1w64_dl1_2k1w32_ul2_32k4w64_DDEI	il1_4k1w64_dl1_2k1w64_ul2_24k3w64_DDEU
epic	il1_2k1w16_dl1_8k1w16_ul2_32k4w16_DIUU	il1_2k1w16_dl1_8k1w16_ul2_32k4w16_DDEU	
fir	il1_4k1w32_dl1_2k1w64_ul2_32k4w64_DDDI	il1_8k1w16_dl1_8k1w16_ul2_32k4w16_DDEU	
g3fax	il1_4k1w64_dl1_2k1w16_ul2_32k4w64_DIUU	il1_4k1w64_dl1_2k1w16_ul2_32k4w64_DDEI	
jpeg	il1_8k1w32_dl1_8k1w16_ul2_32k4w32_DDDI	il1_8k1w16_dl1_8k1w16_ul2_32k4w16_DDEU	
matmul	il1_2k1w16_dl1_2k1w64_ul2_32k4w64_DEEI	il1_2k1w16_dl1_8k2w16_ul2_32k4w16_DDEU	
mpeg2	il1_4k1w32_dl1_4k1w16_ul2_32k4w32_DIUU	il1_4k1w16_dl1_4k2w16_ul2_32k4w16_DDEU	
pocsag	il1_8k1w64_dl1_4k1w64_ul2_32k4w64_DIUU	il1_8k1w16_dl1_8k1w16_ul2_32k4w16_DEIU	
ps-jpeg	il1_4k1w64_dl1_2k1w16_ul2_32k4w64_DIIU	il1_4k1w16_dl1_2k1w16_ul2_32k4w16_DEIU	
ucbqsort	il1_4k1w64_dl1_4k1w64_ul2_32k4w64_DIUU	il1_2k1w16_dl1_2k1w16_ul2_32k4w16_DEIU	
v42	il1_8k1w64_dl1_8k1w16_ul2_32k4w64_DIUU	il1_8k1w32_dl1_4k1w16_ul2_32k4w32_DDEU	
pegwit	il1_8k1w64_dl1_8k1w16_ul2_32k4w64_DDDI	il1_8k1w16_dl1_4k2w16_ul2_32k4w16_EEEU	
g721	il1_8k1w16_dl1_2k1w16_ul2_32k4w16_DDDI	il1_8k1w16_dl1_2k1w16_ul2_32k4w16_EEEU	
rawcaudio	il1_2k1w64_dl1_2k1w16_ul2_32k4w64_DIUU	il1_2k1w16_dl1_2k1w16_ul2_32k4w16_DDEI	

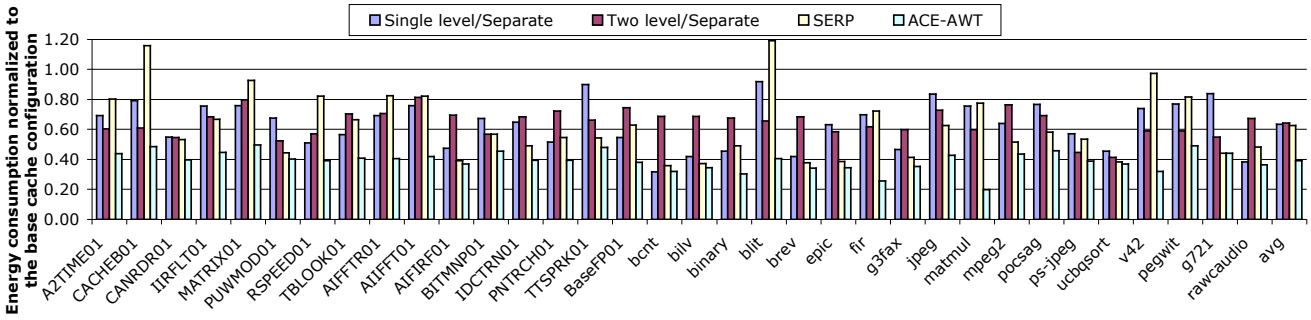

Fig. 12. Energy consumption of various heuristic configurations normalized to the base cache configuration for each heuristics base cache hierarchy.

that nearly achieve the optimal energy savings.

As expected, the heuristic configurations and the optimal cache configurations vary greatly across all benchmarks. However, we observe a common trend in level two configurations. Due to the statistically higher miss rates in level one data caches compared to level one instruction caches, the level two cache tends to devote more ways to caching data than caching instructions. In future work, this observation could be exploited to further refine the ACE-AWT fine tuning phase.

E. Comparing to Previous Heuristics

We compare both SERP and AWE-ACT to previous state-of-the-art heuristics for configuration a single level of cache [25] and two levels of cache with separate instruction and data caches for the second level [12]. Fig. 12 shows energy consumption normalized to the base cache configuration for each heuristic. On average, heuristics for a single level of cache, a two level cache with separate second level caches, and SERP perform nearly the same with average energy savings of nearly 40%. Looking at individual benchmarks, we see varying behavior for these three heuristics with SERP

TABLE II
NUMBER OF CONFIGURATIONS EXPLORED FOR VARIOUS CACHE TUNING HEURISTICS.

	Single level/Separate	Two level/Separate	SERP	ACE-AWT
range	7-13	28-28	22-24	23-41
average	9.8	28	23.8	33.7

outperforming both previous cache tuning heuristics in half of the benchmarks. However, ACE-AWT consistently outperforms both previous heuristics in all benchmarks.

TABLE II compares the number of cache configurations explored by SERP and ACE-AWT to previous methods. The heuristic for a single level of cache only explores 9.8 configurations on average and results in the same average energy savings as both the two level/separate and SERP heuristics. When comparing SERP with two-level/separate, SERP achieves the same average energy savings and explores four fewer cache configurations. ACE-AWT explores 33.7 configurations on average, 3.4 times that of the single level cache, but achieves 61% energy savings as compared to only 40% energy savings.

F. Static Energy for Future Trends

For the results presented in section B, 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 to 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 technologies, 50% of the total energy consumption of the cache.

TABLE III shows the average energy consumption normalized to the base cache configuration averaged across all benchmarks for the heuristics studied. Energy consumptions that show ***energy savings*** are highlighted 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, SERP 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 III 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.

TABLE III
ENERGY CONSUMPTION NORMALIZED TO THE BASE CACHE CONFIGURATION AVERAGED ACROSS ALL BENCHMARKS FOR DIFFERENT STATIC ENERGY CONSUMPTION. ENERGY SAVINGS ARE SHOWN IN BOLD.

	SERP	ACE-AWT
10% Static Energy	1.24	**0.38**
15% Static Energy	1.18	**0.37**
20% Static Energy	1.10	**0.33**
25% Static Energy	1.05	**0.32**
50% Static Energy	**0.82**	**0.23**

VI. 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. [26] 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 did not have any overhead.

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 [16][21]. To better accommodate a single application environment with multiple phase changes, the tuning hardware could 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 components 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 would 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.

VII. 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 61% 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 9% due to line size configuration.

Future work includes recompilation of the application to the best cache configuration for further energy and performance benefits. We also plan to examine desktop and mainframe applications on appropriate cache configurations for different application execution phases and verify that the heuristic developed in this work is applicable to desktop applications exhibiting different access pattern characteristics than embedded applications. Additionally, we plan to explore the many details involved with runtime implementation of application phase-based cache tuning. Furthermore, we plan to generalize our tuning heuristic so that it is amenable to future platforms with an unknown number of levels within the cache hierarchy.

References

[1] D.H. Albonesi, "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.

[3] Arc International, http://www.arccores.com.

[4] ARM, http://www.arm.com.

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

[6] D. Burger, T. Austin, and S. Bennet, "Evaluating future microprocessors: the simplescalar toolset." *University of Wisconsin-Madison Computer Science Department Tech. Report CS-TR-1308*, July 2000.

[7] A. S. Dhodapkar and J. E. Smith, "Timing reconfigurable microarchitectures for power efficiency." *International Symposium on Parallel and Distributed Processing*, 2004

[8] EEMBC, The Embedded Microprocessor Benchmark Consortium, http://www.eembc.org.

[9] Z. Ge, H. B. Lim, and W. F. Wong, "A reconfigurable instruction memory hierarchy for embedded systems." *International Conference on Field Programmable Logic and Applications*, 2005.

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

[11] A. Ghosh and T. Givargis, "Cache optimization for embedded processor cores: an analytical approach." *International Conference on Computer Aided Design*, November 2003.

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

[13] A. Gordon-Ross, F. Vahid, and N. Dutt, "Fast configurable cache tuning with a unified second level cache." *International Symposium on Low Power Electronic Design,* August 2005.

[14] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems." *International Symposium on Microarchitecture*, 1997.

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

[16] M.C. Merten, A.R. Trick, C.N. George, J. Gyllenhaal, and W.W. Hwu, "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.

[17] MIPS Technologies, http://www.mips.com.

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

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

[20] S. Segars, "Low power design techniques for microprocessors." *International Solid State Circuit Conference*, February 2001.

[21] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and exploiting program phases." *IEEE Micro: Micro's Top Picks from Computer Architecture Conferences*, December 2003

[22] Tensilica, Xtensa Processor Generator, http://www.tensilica.com/.

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

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

[25] C. Zhang, F. Vahid, and R. Lysecky, "A self-tuning cache architecture for embedded systems." *Special Issue on Dynamically Adaptable*

Embedded System, ACM Transactions on Embedded Computing Systems Vol 3, No 2, May 2004, Pages 1-19.

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

Ann Gordon-Ross is an Assistant Professor of Electrical and Computer Engineering at the University of Florida. She received her B.S. and Ph.D. in Computer Science from the University of California, Riverside in 2000 and 2006, respectively. She is a member of the NSF Center for High-Performance Reconfigurable Computing (CHREC) at the University of Florida.

Her research interests include embedded systems, computer engineering, low-power design, reconfigurable computing, platform design, dynamic optimizations, hardware design, real-time systems, computer architecture, and multi-core platforms

Frank Vahid is a Professor of Computer Science and Engineering at the University of California, Riverside, and Associate Director of the Center for Embedded Computer Systems at UC Irvine. He received a B.S. in Computer Engineering from the University of Illinois in 1988, and M.S. and Ph.D. degrees from the University of California, Irvine in 1990 and 1994, respectively.

He is author of the textbooks Digital Design (John Wiley and Sons, 2006), VHDL for Digital Design (John Wiley and Sons, 2007), Verilog for Digital Design (John Wiley and Sons, 2007), and Embedded System Design (John Wiley and Sons, 2001).

Nikil D. Dutt received a Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 1989, and is currently a Chancellor's Professor at the University of California, Irvine, with academic appointments in the CS and EECS departments.

His research interests are in embedded systems, electronic design automation, computer architecture, optimizing compilers, system specification techniques, and distributed embedded systems.

He received best paper awards at CHDL89, CHDL91, VLSIDesign2003, CODES+ISSS 2003, CNCC 2006, and ASPDAC-2006. Professor Dutt currently serves as Editor-in-Chief of ACM Transactions on Design Automation of Electronic Systems (TODAES) and as Associate Editor of ACM Transactions on Embedded Computer Systems (TECS) and of IEEE Transactions on VLSI Systems (IEEE T-VLSI). He was an ACM SIGDA Distinguished Lecturer during 2001-2002 and an IEEE Computer Society Distinguished Visitor for 2003-2005. He has served on the steering, organizing, and program committees of several premier CAD and Embedded System Design conferences and workshops, including ASPDAC, CASES, CODES+ISSS, DATE, ICCAD, ISLPED and LCTES. He serves or has served on the advisory boards of ACM SIGBED and ACM SIGDA, and previously served as Vice-Chair of ACM SIGDA and of IFIP WG 10.5. Professor Dutt is a Fellow of the IEEE, an ACM Distinguished Scientist, and an IFIP Silver Core Awardee.