Automatic Tuning of Two-Level Caches to Embedded Applications

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

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

Abstract

The power consumed by the memory hierarchy of a microprocessor can contribute to as much as 50% of the total microprocessor system power, and is thus a good candidate for optimizations. We present an automated method for tuning two-level caches to embedded applications for reduced energy consumption. The method is applicable to both a simulation-based exploration environment and a hardware-based system prototyping environment. We introduce the two-level cache tuner*, or* TCaT - *a heuristic for searching the huge solution space of possible configurations. The heuristic interlaces the exploration of the two cache levels and searches the various cache parameters in a specific order based on their impact on energy. We show the integrity of our heuristic across multiple memory configurations and even in the presence of hardware/software partitioning – a common optimization capable of achieving significant speedups and/or reduced energy consumption. We apply our exploration heuristic to a large set of embedded applications. Our experiments demonstrate the efficacy of our heuristic: on average the heuristic examines only 7% of the possible cache configurations, but results in cache sub-system energy savings of 55%, only 1% more than the optimal cache configuration. In addition, the configured cache achieves an average speedup of 28% over the base cache configuration due to tuning of cache line size to the application's needs.*

Keywords

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

1. Introduction

The power consumed by the memory hierarchy of a microprocessor can contribute to as much as 50% of the total microprocessor system power [16]. Such a large contributor to power is a good candidate for optimizations.

Tuning cache parameters to the needs of a particular application can save energy. Every application has different cache requirements that cannot be efficiently satisfied with one predetermined cache configuration. Miss rates or energy consumption may be unnecessarily high if a cache configuration does not fit the application. For instance, the cache size should reflect the working set of the application. If the cache is too large, the energy required to fetch from the larger cache may be unnecessarily high. If the cache is too small, excess energy may be wasted due to thrashing. Furthermore, an application may have a large or small amount of spatial locality, favoring either a large or small line size, respectively. Excess energy may be consumed during stall cycles if the line size is too small, but energy may be wasted fetching unused information from main memory if the line size is too large. Additionally, cache associativity should reflect the application's temporal locality. The diversity among applications leads to very different cache requirements for different applications [21].

New technologies enable tuning of cache parameters to the needs of an application. Core-based processors allow a designer to choose a particular cache configuration [2][3][4][13][18] and there are instances of processor designs that allow their caches to be configured during system reset or even during runtime [1][12][21].

Finding the best cache configuration for an application is difficult. A single-level cache may have dozens of different cache configurations, and interdependent multi-level caches lead to thousands of different cache configurations. The configuration space gets even larger if other dependent configurable architecture parameters are considered, such as bus and processor parameters. Exhaustively searching the space is too slow even if fully automated. However, with possible average energy savings of over 40% for a configurable memory hierarchy [5][21], we believe that cache configuration is worth the effort.

Cache configuration can either be done at design time or in-system during runtime. If cache configuration is done at design time, a simulation method is typically used to search the design space. Simulation-based exploration is particularly costly in terms of time since searching even one configuration may take hours to simulate. Using an exploration heuristic can reduce the number of explored configurations. However, for the simulation to be accurate, the system must be well modeled with realistic input stimuli. Often, setting up a realistic simulation environment may be more difficult than designing the system itself.

The alternative to cache configuration at design time is cache configuration in-system during runtime. Runtime configuration is accomplished with a configurable cache and hardware to drive the cache exploration. While exploration time is greatly reduced due to real-time execution, the exploration itself can interfere with system behavior of many embedded systems. This interference may not be tolerated in real-time systems with strict timing requirements. For in-system exploration to be useful, a relatively non-intrusive heuristic is needed where accuracy may be traded off for reduced intrusion or reduced power during exploration.

Whether cache exploration is done during design time or runtime, an exhaustive approach may not be feasible. Good

heuristics based on typical cache behavior can find near optimal results by searching a fraction of the exploration space. This is possible because of the highly predictable nature of cache and application behavior. Caches are designed to exploit the temporal and spatial locality of applications and good heuristics can exploit this behavior to converge on an optimal cache configuration quickly.

A popular system optimization that greatly affects an application's cache behavior is hardware/software partitioning. Current methods of hardware/software partitioning discover the frequent loops of an application and convert these frequent loops into hardware running on a field programmable gate array (FPGA) for reduced energy consumption and improved performance (e.g., [17]). The frequent loops can amount to as much as 90% or more of the execution time. Removing these frequent loops from software alters cache behavior greatly by removing spatial and temporal locality from the application. An application's locality comes mainly from highly iterated loops, however these loops are now implemented in hardware. In all hardware/software platforms, the implementation of frequent loops in the FPGA affects instruction cache behavior because the instructions associated with these loops are no longer fetched and stored in the instruction cache. The data cache will also be affected in systems where the FPGA does not access the cache structures. We observed that some heuristics that worked well for non-partitioned systems, did not work well after the system is partitioned because the removal of locality from the application by hardware/software partitioning causes cache behavior to become unpredictable. We therefore consider both non-partitioned and partitioned systems in our cache configuration method.

In this paper, we discuss related work on automated cache configuration both during design time and in-system during runtime. We describe *the two-level cache tuner*, or *TCaT* - a heuristic for automatically searching a configurable two-level cache hierarchy with configurable size, associativity, and line size. The basic idea of the TCaT is to interlace the exploration of the level one and level two caches, exploring the most important cache parameters first. The TCaT performs well on non-partitioned systems, and also improves greatly upon previous heuristics in the presence of hardware/software partitioning. The TCaT is targeted for design time exploration and may be applied to a simulation environment or a hardware-based prototyping environment.

2. Related Work

Many methods exist for configuring a single level of cache to a particular application during design time and in-system during runtime. Cache configuration can be specified during design time for many commercial soft cores from MIPS [13], ARM [4], and Arc [3] and for environments such as Tensilica's Xtensa processor generator [18] and Altera's Nios embedded processor system [2].

Configurable cache hardware also exists to assist in cache configuration. Motorola's M*CORE [12] processors offer way configuration which allows the ways of a unified data/instruction cache to individually be specified as either data or instruction ways. Additionally, ways may be shut down entirely. Way shut-down is further explored by Albonesi [1] to reduce dynamic power by an average of 40%. An adaptive cache line size methodology is proposed by Veidenbaum et al. [19] to reduce memory traffic by more than 50%. Zhang et al. [21] proposes a methodology called way-concatenation where a cache can be configured by software to either be direct-mapped, 2- or 4-way set associative achieving average energy savings of 40% compared to a conventional 4-way set associative cache.

Exhaustive search methods may be used to find optimal cache configurations, but the time required for an exhaustive search is often prohibitive. Several tools do exist for assisting designers in tuning a single level of cache. 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.

Heuristic methods exist to prune the search space of the configurable cache. Palesi et al. [14] improves upon the exhaustive search used in Platune by using a genetic algorithm to produce comparable results in less time. Zhang et al. [20] presents a cache configuration exploration methodology for prototyping platforms, wherein a cache exploration component searches configurations in order of their impact on energy, and produces a list of Pareto-optimal points representing reasonable tradeoffs in energy and performance. Ghosh et al. [9] uses an analytical model to efficiently explore cache size and associativity and directly computes a cache configuration to meet the designers' performance constraints.

Few methods exist for tuning multiple levels of a cache hierarchy. Balasubramonian et al. [5] proposes a hardware-based cache configuration management algorithm to improve memory hierarchy performance while considering energy consumption. An average reduction in memory hierarchy energy of 43% can be achieved with a configurable level two and level three cache hierarchy coupled with a conventional level one cache.

To the best of our knowledge, no previous work has explored the integrity of cache tuning in the presence of hardware/software partitioning like we do, yet such partitioning is becoming increasingly common in embedded systems, especially with the advent of single-chip microprocessor/FPGA platforms.

3. Configurable Cache Architecture

The configurable caches in each of the two cache levels explored in this paper are based on the configurable cache architecture described for a single level configurable cache by Zhang et al. [21]. The target architecture for our two-level cache tuning heuristic contains separate level one instruction and data caches and separate level two instruction and data caches.

The cache architecture supports a certain range of configurations as detailed in Zhang et al. [21]. The base level-one cache of 8 Kbytes consists of four banks that can operate as four ways. A special configuration register allows the ways to be concatenated to form either a direct-mapped or 2-way set associative 8 Kbyte cache. The configuration register may also be configured to shut down ways, resulting in a 4 Kbyte direct-mapped or 2-way set associative cache or a 2 Kbyte direct-mapped cache. Specifically, 2 Kbyte 2- or 4-way set associative and 4 Kbyte 4-way set associative caches are not possible using the configurable cache hardware. The same limitations apply to the level two cache with the base cache being a 64 Kbyte cache with four banks – each bank is 16 Kbyte and similarly to the 8 Kbyte cache with four banks, a 2-way 16 Kbyte cache is not possible. Although the cache architecture is limited to certain configurations in a hardware-based prototyping environment, our work can be easily extended to

include all possible configurations for a simulation-based environment.

An exhaustive exploration of all cache configurations for a two level cache hierarchy is too costly. For a single level separate instruction and data cache design, an exhaustive exploration would explore a total of 28 different cache configurations. However, the addition of a second level of hierarchy raises the number of cache configurations to 432.

Nevertheless, for comparison purposes, we determined the optimal cache configuration for each benchmark by generating exhaustive data. It took several months of continual simulation time on an UltraSparc compute server to generate the data for our 34 benchmarks.

4. Experimental Environment

For our studies, we used six benchmarks from the MediaBench benchmark suite [11], twelve benchmarks from the Powerstone benchmark suite [12] and sixteen benchmarks for the EEMBC benchmark suite [7] as shown in Table 1. The results for a subset of these benchmarks can be seen in [10]. The benchmarks are small embedded applications geared towards low power embedded systems. We ran all benchmarks on SimpleScalar [6] for each cache configuration to determine the cache hits and misses.

Table 1: Benchmark descriptions.

Benchmark	Description
epic*	Efficient Pyramid Image Coder
g721*	Voice Compression
jpeg*	JPEG Compression
mpeg2*	MPEG Compression
rawcaudio*	Voice Encoding
pegwit*	Public Key Encryption
g3fax**	Group Three Fax Decoding
bcnt**	Bit Manipulation
bilv**	Shift, AND, OR operations
binary**	Binary Insertion
fir**	FIR Filtering
blit**	Graphics Application
brev**	Shifting and Or Operations
matmul**	Matrix Multiplication
pocsag**	Signal Processing
ps-jpeg**	JPEG Compression
ucbqsort**	U.C.B. Quick Sort
v42**	Modem Encoding/Decoding
A2TIME01***	Angle-to-Time Conversion
AIFFTR01***	Fast Fourier Transform
AIFIRF01***	Finite Impulse Response (FIR) Filter
AIIFFT01***	Inverse Fast Fourier Transform
BaseFP01***	Basic Integer and Floating Point
BITMNP01***	Bit Manipulation
CACHEB01***	Cache Buster
CANRDR01***	Response to Remote Request
IDCTRN01***	Inverse Discrete Cosine Transform
IIRFLT01***	Low-Pass Filter and DSP functions
MATRIX01***	Matrix Math
PNTRCH01***	Pointer Chasing
PUWMOD01***	Pulse-Width Modulation
RSPEED01***	Road Speed Calculation
TBLOOK01***	Table Lookup and Interpolation
TTSPRK01***	Tooth To Spark

** MediaBench **Powerstone ***EEMBC*

We determined the energy of the system using both estimation methods and measurements. We obtained the dynamic energy consumed by a cache fetch for each cache configuration using the CACTI [15] model for 0.18-micron technology (in separate work, we compared a cache layout to the CACTI model and found CACTI's estimates to be very close [21]). We obtained the energy consumed by a fetch from main memory from a standard Samsung memory. For static energy consumption of the caches, we assume the static energy accounts for 10% of the total energy of the cache (reasonable for the next few years' technologies). We obtained CPU stall energy from a 0.18-micron MIPS microprocessor.

For the base system, we estimated the cache miss penalties and memory throughput using typical ratios for an embedded system. A fetch from the level one cache is used as the base fetch time for the system and all other miss penalties are derived as ratios of the base. We assume that a fetch from the level two cache takes four times longer than a fetch from level one cache. A fetch from main memory takes ten times longer than a fetch from level two cache. We assign the memory throughput as 50% of the latency, meaning that after the first block of a request is transferred, it takes 50% of the original latency to transfer each remaining block. In Section 6.4, we will explore different system configurations.

We chose the cache parameters to explore based on the benchmarks chosen and to reflect cache configurations of typical embedded systems [21]. We chose cache sizes large enough to be realistic yet small enough so that the entire benchmark did not fit into the cache. For the level one cache, we explore cache sizes of 2, 4, and 8 Kbytes. For the level two cache, we explore cache sizes of 16, 32, and 64 Kbytes. The associativities and line sizes chosen reflect typical off-the-shelf embedded systems. We explore direct mapped, 2-, and 4-way set associativities, and cache line sizes of 16, 32, and 64 bytes. For comparison, we have chosen a **base cache** hierarchy configuration consisting of an 8 Kbyte, 4-way set associative level one cache with a 32 byte line size, and a 64 Kbyte 4-way set associative level two cache with a 64 byte line size – a reasonably common configuration.

We based the hardware/software partitioning of the benchmarks on the method described by Stitt et al. in [17] for a system where the FPGA does not access the cache structures. Stitt determines the frequent loops in an application and partitions the most frequent loops into hardware. We use a loop profiling tool to determine the loop structure and loop frequencies of the benchmarks. We determine the most frequent loops to partition to hardware and modify SimpleScalar so that the instructions and data fetched within the loops are not included in the cache simulation.

5. Initial Two-Level Cache Tuning Heuristic – Searching Each Level Independently

When developing a good heuristic, the parameter (cache size, line size, or associativity) with the largest impact in performance and energy would likely be the best parameter to search first. Zhang et al. [20] showed the importance of each parameter by holding two parameters steady and varying the third to study the impact that the varied parameter had on miss rates and energy consumption. Zhang concluded that the most

important cache parameter is cache size, followed by line size, and finally associativity.

The heuristic developed by Zhang et al., based on the importance of the cache parameters, is summarized below:

(1) Begin with a 2 Kbyte, direct-mapped cache with a 16 byte line size. Increase the cache size to 4 Kbytes. If the increase in cache size causes a decrease in energy consumption, increase the cache size to 8 Kbytes. Choose the cache size with the best energy consumption.
(2) For the best cache size determined in step (1), increase the line size from 16 bytes to 32 bytes. If the increase in line size causes a decrease in energy consumption, increase the line size to 64 bytes. Choose the line size with the best energy consumption.
(3) For the best cache size determined in step (1) and the best line size determined in step (2), increase the associativity to 2 ways. If the increase in associativity causes a decrease in energy consumption, increase the associativity to 4 ways. Choose the associativity with the best energy consumption.

The heuristic described was developed for a single level of cache. We initially extended this heuristic to a two level hierarchy by exploring the level one cache while holding the level two cache at the smallest size. Once the level one cache is configured, the level two cache is explored using the same heuristic.

We applied the initial heuristic to the benchmarks and found that this heuristic did not perform well for two levels (the original heuristic was intended for only one level, for which the heuristic works well). The cache configuration determined by our initial heuristic consumed, on average over all benchmarks for, 1.65 times more energy than the optimal configuration for a non-partitioned system, and 1.59 times more energy than the optimal configuration for a hardware/software partitioned system. In the worst case, our initial heuristic found a cache configuration that had 3.72 times more energy consumption than the optimal configuration.

The naïve assumption that the two levels of cache could be configured independently was the reason that our initial heuristic did not perform well for a two level system. In a two level cache hierarchy, the behavior of each cache level directly affects the behavior of the other level. For example, the miss rate of the level one cache does not solely determine the performance of the level two cache. The performance of the level two cache is also determined by what *values* are missing in the level one cache. To fully explore the dependencies between the two levels, we decided to explore both levels simultaneously.

6. Interlaced Heuristic

To more fully explore the dependencies between the two levels, we expanded our initial heuristic to interlace the exploration of the level one and level two caches. Instead of entirely configuring the level one cache before configuring the level two cache, the interlaced heuristic explores one parameter for both levels of cache before exploring the next parameter. The basic intuition behind our heuristic is that interlacing the exploration allows for better modeling and tuning of the interdependencies between the different levels of cache hierarchy.

6.1 Basic Interlacing

The interlaced heuristic follows the same parameter exploration ordering as the initial heuristic. However, instead of exploring the cache configuration one level at a time, the interlaced heuristic explores one parameter at a time: the optimal size is found for the level one cache, followed by the optimal size for the level two cache. Then, the optimal line size is found for the level one cache, followed by the optimal line size for the level two cache. Finally, the optimal associativity is found for the level one cache, followed by the optimal associativity for the level two cache.

We applied the interlaced heuristic to the benchmarks and found that the interlaced heuristic performed much better than the initial heuristic, but there was still much room for improvement.

6.2 Interlaced Heuristic with Full Parameter Exploration

We examined the cases where the interlaced heuristic did not yield the optimal solution. We discovered that in these cases, the optimal was not being reached because the initial heuristic did not fully explore each parameter. For instance, if an increase from a 2 Kbyte to 4 Kbyte cache size did not yield an improvement in energy, an 8 Kbyte cache size was not examined. For a system with only a level one cache, not fully exploring each parameter had little to no adverse impact on the quality of results produced by the initial heuristic. However, when exploring two levels of caches, the limited exploration of each parameter does not allow the dependencies between the two caches to be explored fully. Furthermore, in a hardware/software partitioned system, the limited parameter exploration did not allow the unpredictable cache behavior to be fully explored.

To improve upon the interlaced heuristic, we added full parameter exploration, meaning the heuristic checks all possible values for a given parameter. We applied the interlaced heuristic with full parameter exploration to the benchmarks. The results were much improved over the previous attempt. However, a few benchmarks still performed poorly.

6.3 The Improved Interlaced Heuristic with Full Parameter Exploration and Final Adjustment – TCaT

We examined the cases where the interlaced heuristic with full parameter exploration was far from optimal. We determined that the reason for being far from optimal was not due to a failure in the heuristic, but rather due to the limitations set on certain cache configurations by the configurable cache itself. For example, in the level two cache, if a 16 Kbyte cache is chosen as the best size, the only associativity available is a direct-mapped cache. With no energy improvement by increasing the cache from 16 Kbyte direct-mapped to a 32 Kbyte direct-mapped cache, no other associativities are searched by the previous heuristics. In some cases, the optimal cache was indeed a 32 Kbyte 2-way set associative cache. To allow for all associativities to be searched, we added a final adjustment to the associativity search step of the interlaced heuristic with full parameter exploration. The final adjustment allows the cache size to be increased for both the level one and level two caches in order to search larger associativities. We refer to this final heuristic as the two-level cache tuner - the TCaT.

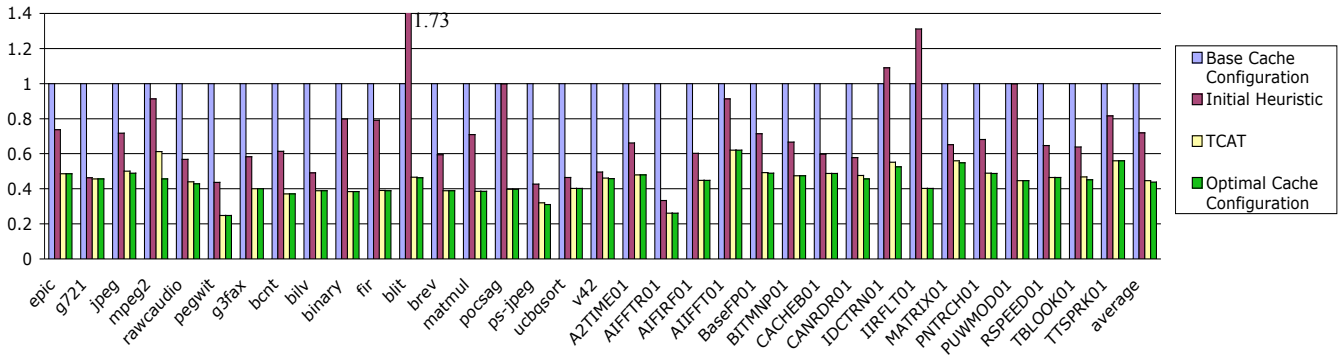


Figure 1: Energy consumption for the initial heuristic cache configuration, the TCaT cache configuration, and the optimal cache configuration normalized to the base cache configuration for each benchmark for a non-hardware/software partitioned system.

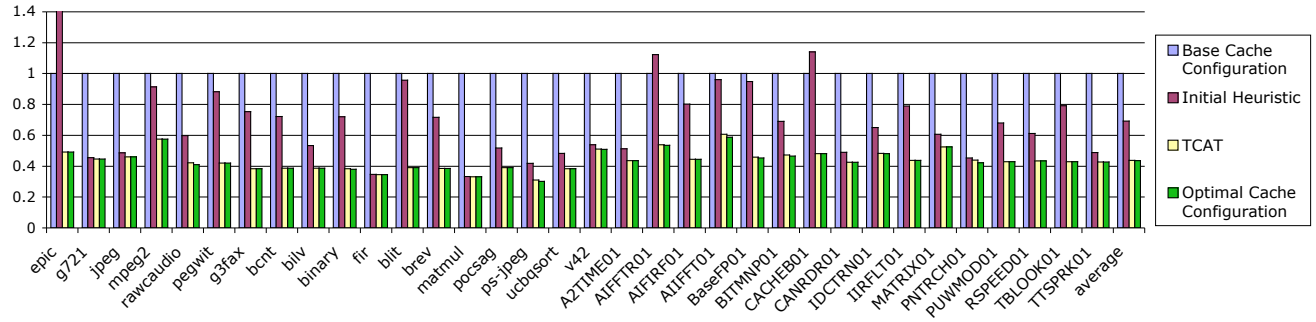


Figure 2: Energy consumption for the initial heuristic cache configuration, the TCaT cache configuration, and the optimal cache configuration normalized to the base cache configuration for each benchmark for a hardware/software partitioned system.

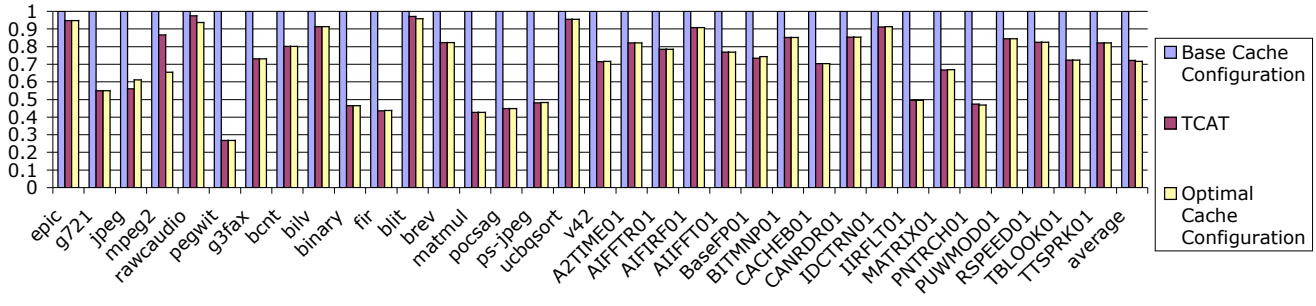
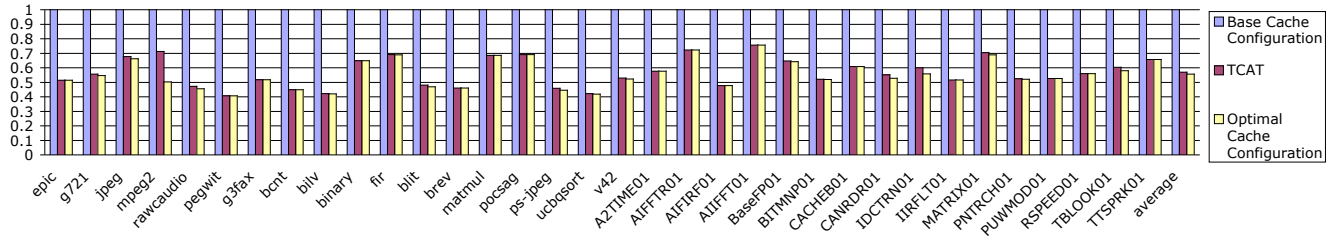


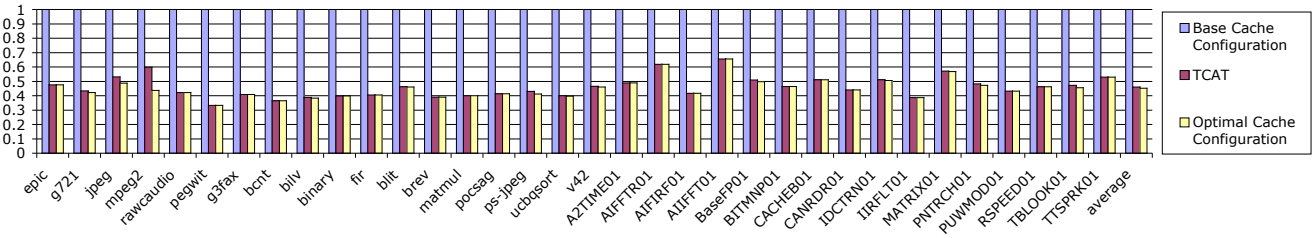
Figure 3: Execution time of the benchmarks for the TCaT cache configuration and the optimal cache configuration (normalized to the execution time of the benchmark running with the base cache configuration).

Figure 1 shows the results for the initial heuristic and the TCaT for each benchmark in a non-partitioned system. The energy consumptions have been normalized to the base cache configuration for each benchmark's instruction and data caches. The results show that the TCaT finds the optimal cache configuration in most cases. Compared to the base cache configuration and averaged over all benchmarks, the initial heuristic achieves an average energy savings of 28% while the TCaT achieves an average energy savings of 55%. On average, the TCaT finds a cache configuration with energy consumption only 0.8% higher than the optimal cache configuration. In a number of benchmarks such as blit, pocsag, and IIRFLT01, the TCaT greatly outperforms the initial heuristic. In these benchmarks, the initial heuristic finds cache configurations that are higher in energy than the base cache configurations, while the TCaT finds the optimal cache configurations.

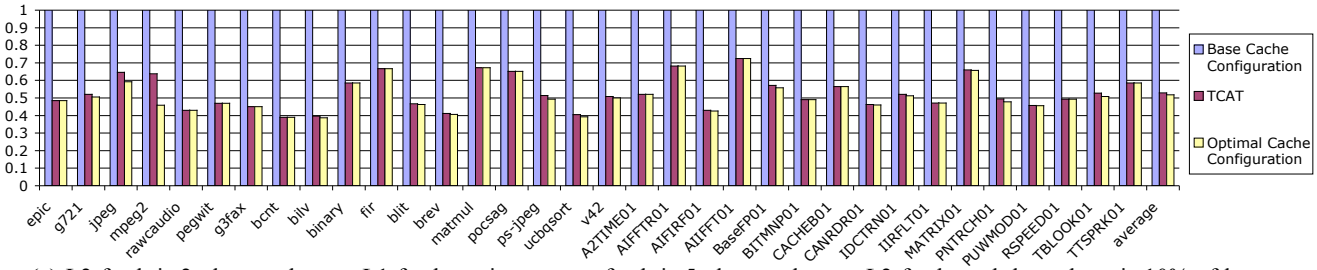
Figure 2 shows the results for the initial heuristic and the TCaT for each benchmark in a partitioned system. The energy consumptions have been normalized to the base cache configuration for each benchmark's instruction and data caches. The results show the integrity of the TCaT in the presence of hardware/software partitioning. On average, the TCaT finds a cache configuration that has an energy consumption only 0.02% higher than the optimal cache



(a) L2 fetch is 4x longer than an L1 fetch, main memory fetch is 10x longer than an L2 fetch, and throughput is 10% of latency



(b) L2 fetch is 2x longer than an L1 fetch, main memory fetch is 5x longer than an L2 fetch, and throughput is 50% of latency



(c) L2 fetch is 2x longer than an L1 fetch, main memory fetch is 5x longer than an L2 fetch, and throughput is 10% of latency

Figure 4: Energy consumption for the TCaT cache configuration and the optimal cache configuration normalized to the base cache configuration for a non-partitioned system using different system memory configurations

configuration, translating to an energy savings of 56% over the base cache configuration.

The TCaT focuses on determining the optimal cache configuration with respect to energy consumption. However, it is important to verify that the performance of the system is not adversely affected. Figure 3 shows the execution times for each benchmark for the TCaT cache configuration and the optimal cache configuration normalized to the execution time for the base cache configuration. Figure 3 represents a system without hardware/software partitioning. The performance of the system with hardware/software partitioning is highly dependent on the partitioning itself, the speed of the processor and the speed of the configurable fabric used. Details on the performance estimation methods of the hardware/software partitioning is beyond the scope of this paper.

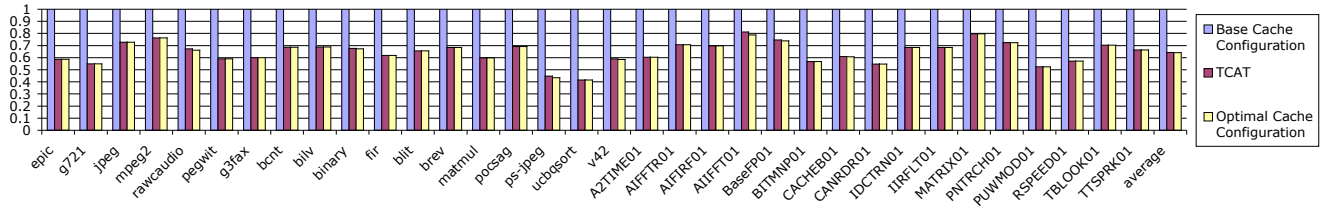
Figure 3 shows that for every benchmark, there is no loss of performance due to cache configuration for optimal energy consumption. In fact, the benchmarks receive an average of a 28% speedup due to the optimal configuration of the cache line size.

6.4 Integrity of the TCaT Across Different Configurations

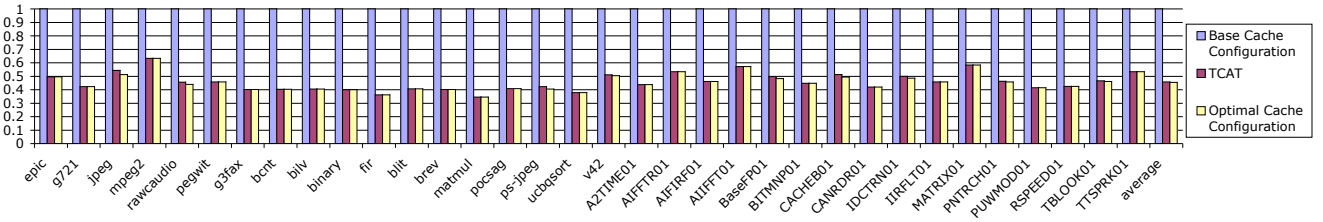
We studied the integrity of the TCaT across three additional system configurations. For a non-partitioned system, Figure 4(a) shows the results for a system with a fetch from the level two cache being four times longer than a fetch from the level one cache, with a fetch from main memory being ten times longer than a fetch from the level two cache and a memory throughput of 10% of the miss latency. Figure 4(b) and Figure 4(c) both show the results for a system with a fetch from the level two cache being two times longer than a fetch from the level one cache and with a fetch from main memory being five times longer than a fetch from the level two cache. The memory throughput for Figure 4(b) and Figure 4(c) is 50% and 10% of the miss latency respectively.

For all of the different system configurations, the average energy consumption over all benchmarks of the TCaT configurations consumed on average only 1% more energy than the optimal cache configurations. The TCaT shows good integrity over all benchmarks regardless of the system configuration.

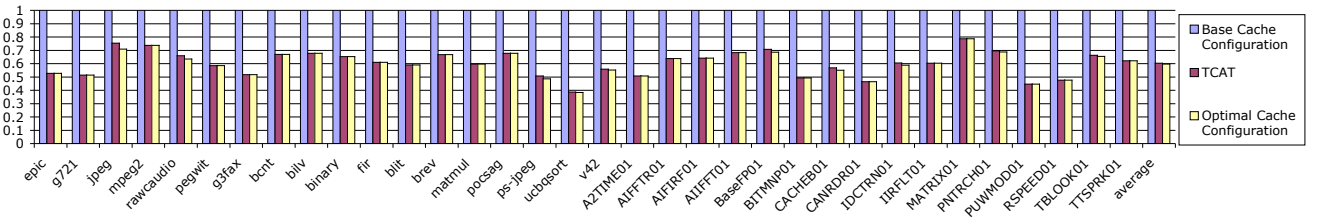
Figure 5 shows the integrity of the TCaT over the same three additional system configurations in the presence of hardware/software partitioning. For all of the different system configurations, the average energy consumption over all benchmarks of the TCaT configurations consumed on average less than 1% more energy than the optimal cache configurations. The TCaT shows good integrity in the presence of hardware/software partitioning regardless of the system configuration,



(a) L2 fetch is 4x longer than an L1 fetch, main memory fetch is 10x longer than an L2 fetch, and throughput is 10% of latency



(b) L2 fetch is 2x longer than an L1 fetch, main memory fetch is 5x longer than an L2 fetch, and throughput is 50% of latency



(c) L2 fetch is 2x longer than an L1 fetch, main memory fetch is 5x longer than an L2 fetch, and throughput is 10% of latency

Figure 5: Energy consumption for the TCaT cache configuration and the optimal cache configuration normalized to the base cache configuration for a hardware/software partitioned system using different system memory configurations

6.5 TCaT Exploration Time

The TCaT reduces the configuration search space significantly. The exhaustive approach for separate instruction and data caches for a two level cache hierarchy explores 432 cache configurations. The improved heuristic explores 28 cache configurations, or 6.5% of the search space. The reduction speeds up both a simulation approach and a hardware-based prototyping platform approach.

For a simulation-based approach, we examined the exploration time in hours for the exhaustive and TCaT approaches for cache configuration exploration on a Sun workstation running SunOS 5.8 with dual 500 MHz processors and 2 gigabytes of memory. We determined the execution time for one configuration running on SimpleScalar using the Unix time command. On average over all benchmarks, using the TCaT reduced the exploration time from 25.4 hours down to 1.6 hours, with the longest exploration time being reduced from 204 hours down to 13 hours.

For the hardware-based prototyping environment, we examined the exploration time in seconds for the exhaustive and TCaT approaches for cache configuration exploration on a hypothetical 200 MHz hardware-based configurable prototyping platform. To determine the execution time on the platform, we obtained the total number of instructions executed in the application from SimpleScalar and assumed a CPI of 1.5 for all benchmarks. On average over all benchmarks, using the TCaT reduced the exploration time from 240 seconds down to 16 seconds, with the longest exploration time being reduced from 1811 seconds down to 117 seconds. Keep in mind that the benchmarks used were chosen to simulate in a reasonable amount of time to allow for verification of our heuristic. Larger benchmarks, and more comprehensive input vectors, could take hours to search exhaustively in hardware.

7. Conclusions and Future Work

In this paper, we presented an automated method for tuning two level caches to embedded applications for reduced energy consumption. On average, the TCaT finds a cache configuration that consumes only 1% more energy than the optimal cache configuration for non-hardware/software partitioned applications, and less than 0.1% more energy for hardware/software partitioned applications. The TCaT searches only 6.5% of the design space, translating to a 15 times speedup in exploration time. The TCaT achieves an average energy savings of 55% for a non-hardware/software partitioned system and 56% for a hardware/software partitioned system, with improved performance. We showed the integrity of the TCaT across multiple system configurations. Future work includes extending the TCaT to explore a unified level two cache, recompiling an application after the optimal cache configuration is determined, and performing the heuristic dynamically and transparently during runtime.

8. Acknowledgements

This research was supported in part by the National Science Foundation (CCR-0203829, CCR-9876006) and a Dept. of Education GAANN fellowship, and the Semiconductor Research Corporation (grant CSR 2002-RJ-1046G).

9. References

[1] Albonesi, 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

[3] Arc International, www.arccores.com.

[4] ARM, www.arm.com.

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

[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.

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

[9] Gnosh, 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 in Europe Conference (DATE), February 2004

[11] Lee, C., Potkonjak, M., Mangione-Smith, W.H. MediaBench: a tool for evaluating and synthesizing multimedia and communication systems. Proc 30th Annual International Symposium on Microarchitecture, December 1997.

[12] 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.

[13] MIPS Technologies, 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] Reinman, G., Jouppi, N.P. CACTI2.0: an integrated cache timing and power model. COMPAQ Western Research Lab, 1999.

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

[17] Stitt, G., Vahid, F. The energy advantages of microprocessor platforms with on-chip configurable logic. IEEE Design and Test of Computers, Nov/Dec 2002.

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

[19] Veidenbaum, A., Tang, W., Gupta, R., Nicolau, A., Ji, X. Cache access and cache time model. IEEE Journal of Solid-State Circuits, Vol 31, No 5, 1996.

[20] Zhang, C., Vahid, F. Cache configuration exploration on prototyping platforms. 14th IEEE International Workshop on Rapid System Prototyping , June 2003.

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