A One-Shot Configurable-Cache Tuner for Improved Energy and Performance

Ann Gordon-Ross[1], Pablo Viana[2], Frank Vahid[1,3], Walid Najjar[1], Edna Barros[4]
[1] Department of Computer Science and Engineering – University of California, Riverside
http://www.cs.ucr.edu/~{ann, vahid, najjar}; {ann, vahid, najjar}@cs.ucr.edu
[2]Universidade Federal de Alagoas – Arapiraca-AL, Brazil
[3]Also with the center for Embedded Computer Systems – University of California, Irvine
[4]Centro de Informática – Federal University of Pernambuco, Recife-PE, Brazil

Abstract

We introduce a new non-intrusive on-chip cache-tuning hardware module capable of accurately predicting the best configuration of a configurable cache for an executing application. Previous dynamic cache tuning approaches change the cache configuration several times as part of the tuning search process, executing the application using inferior configurations and temporarily causing energy and performance overhead. The introduced tuner uses a different approach, which non-intrusively collects data on addresses issued by the microprocessor, analyzes that data to predict the best cache configuration, and then updates the cache to the new best configuration in "one-shot," without ever having to examine inferior configurations. The result is less energy and less performance overhead, meaning that cache tuning can be applied more frequently. We show through experiments that the one-shot cache tuner can reduce memory-access related energy for instructions by 35% and comes within 4% of a previous intrusive approach, and results in 4.6 times less energy overhead and a 7.7 times speedup in tuning time compared to a previous intrusive approach, at the main expense of 12% larger size.

1. Introduction

Cache subsystems contribute to a significant percentage of microprocessor system energy consumption, often 50% or more [21]. Caches contain several design parameters, including total size, line size, and associativity. Microprocessor designers typically optimize cache parameter values to achieve good performance/energy across an entire domain of applications, but any one application will usually exhibit better performance/energy if the parameter values could be customized for that application's memory usage characteristics. Previous research shows savings as high as 53% for memory-access-related energy and a 30% performance improvement due to such cache tuning [10].

Due to increasing demand for low-energy microprocessor systems, from small embedded systems seeking to extend battery lifetime to large server systems seeking to reduce electricity and cooling costs, caches with configurable parameter values have been introduced in recent years. Synthesizable (soft-core) processors include numerous parameters whose values can be set by a designer before synthesis [3][17][26]. Albonesi [2], Zhang [28], and Balasubramonian [5] each introduced hardware (hard-core) caches whose parameter values could be adjusted, statically or dynamically, just by setting bits in a configuration register. The commercially-available M*CORE processor [15] had a hardware configurable 4-way cache where each way could be shut down, or configured for instruction, data, or both.

Configurable caches may have many thousands of possible configurations, making tuning to an application a hard problem. Thus, several researchers have proposed automated cache tuning approaches. Most of those approaches assume that tuning is done statically, meaning done once during application design time. Givargis proposed an exhaustive simulation approach [9], while Palesi [18] improved that approach using a genetic search algorithm. Ghosh [8] presented a heuristic that, through an analytical model, directly determined the cache configuration based on the designer's performance constraints and application characteristics.

Other cache tuning approaches can be used dynamically, while an application executes on a microprocessor (requiring of course a dynamically adjustable configurable cache). Zhang [28] presented a single-level cache tuning heuristic that examined on average 5 of 18 possible configurations, taking care to minimize cache flushing for suitability as a dynamic tuning heuristic. Gordon-Ross [11] extended Zhang's heuristic for a two-level configurable cache with 18,000 possible configurations. That heuristic searched only 30 configurations while obtaining 62% energy savings – within 1% of optimal.

Previous dynamic cache tuning approaches search for the best cache configuration for a running application by dynamically adjusting the configurable cache to examine candidate configurations. Examining inferior candidates is intrusive and temporarily introduces energy and/or performance overhead. Thus, those tuning approaches should be applied sparingly so that the overhead of the tuning process does not dominate over the improvements of the tuned cache.

We therefore sought to develop a *non-intrusive* cache tuning approach that would not examine inferior solutions. The idea would be to create a hardware cache-tuning module that non-intrusively monitors an application's memory access patterns and analytically predicts the best cache configuration for those patterns. If the predicted best cache configuration differed from the configuration presently in use (beyond some threshold), the cache tuner would reconfigure the cache directly to the new best cache, i.e., in "one shot".

A one-shot cache tuning approach involves two key challenges. The first is to create an accurate predictor of the best configuration. The second is to keep the size and energy of the one-shot tuning module, which will certainly be more than in previous approaches, down to acceptable values.

The contribution of this paper is the development of the first non-intrusive one-shot cache tuner suitable for hardware implementation. The one-shot tuner is based on a recent cache analysis technique originally developed for simulation-based single-pass multi-cache evaluation. Through modifications of this technique and use of parallel hardware, we show that the one-shot tuner achieves good prediction accuracy and consumes reasonable size and power.

The paper is organized as follows. Section 2 discusses previous research in simultaneous multi-cache evaluation and presents the software technique that we will be extending. Section 3 details the modifications necessary to allow the software technique to be implemented in custom hardware as a one-shot tuner. Section 4 compares our one-shot tuner to a state-of-the-art intrusive cache tuning heuristic. Section 5 gives a practical illustration of the one-shot tuner compared to an intrusive cache tuning heuristic.

2. Simultaneous multi-cache evaluation

2.1 Multiple-pass multi-cache evaluation

Most previous research in cache configuration emphasized fast offline evaluation of multiple cache configurations during simulation. Because simulation is slow, much attention was given to evaluating more than one cache configuration per simulation, i.e., simultaneous multi-cache evaluation. Early research in simultaneous multi-cache evaluation by Mattson el al. [16] defined the inclusion property of caches. The inclusion property states that at any time, the contents of a cache are a subset of the contents of a larger cache, allowing simultaneous evaluation of fully-associative caches of varying sizes using a stack-based methodology. Later, Hill and Smith [12] identified the set refinement property and extended the inclusion property to include direct-mapped and set-associative caches. The set-refinement property observes that blocks that are mapped to the same set in larger caches are also mapped to the same set in smaller caches if the replacement algorithm is a stack algorithm such as the least recently used. Much research utilizes these properties to develop efficient algorithms for simultaneous multi-cache evaluation [6][19][22]. Currently, methods only exist for a single level of cache.

In seeking to develop a one-shot cache tuner, the best approach seemed to be to adapt previous multi-cache evaluation methods, originally intended for software implementation, to a hardware implementation. However, while those methods can simultaneously evaluate varying values for cache parameters including total size, line size, and associativity, the parameters themselves must be explored separately, requiring one exploration pass per parameter, thus requiring multiple passes. Furthermore, some used relatively complex data structures that would be difficult to convert to hardware.

2.2 Single-pass multi-cache evaluation (SPCE)

Recently, however, we presented a technique [27] that proposed a method to evaluate all values for all cache parameters simultaneously, requiring only one simulation pass, and using tables rather than complex data structures. We provide a brief summary here. Like previous simultaneous multi-cache evaluation methods, SPCE (pronounced spee-cee) uses a stack. In addition, a multi-layered table structure is utilized to record cache hits from conflict information gathered from a sequence of addresses.

SPCE uses a stack size equal to the number of static instructions in the application to store the access trace. When an address is processed, the stack is scanned to determine if and when that address was last accessed. If the address is not in the stack, the address represents a new access and the address is pushed to the top of the stack. If the address is in the stack, then all addresses preceding that address in the stack will suggest particular cache configurations that would have yielded a cache hit. After processing the address, the current address is moved to the top of the stack.

Figure 1 illustrates the algorithm that scans the stack to determine which cache configurations would yield a cache hit. *smin, smax, bmin,* and *bmax* refer respectively to the minimum and maximum number of sets and words per line. *amax* refers to the maximum associativity explored. For details, we refer the reader to [27].

SPCE uses a multi-layered table structure to tally cache hits. The number of different assocativities explored determines the number of layers in the table – one layer per associativity. The *update_table* function takes the cache parameter values and updates the appropriate table layer and location.

After processing every address, the tables are used to determine the number of hits for each cache configuration. To verify correct cache hit predictions, we simulated each benchmark studied for every cache configuration. We concluded that SPCE correctly calculates the hit rates for every configuration.

We compared SPCE to a state-of-the-art heuristic for single-level configurable cache exploration and determined that SPCE could produce slightly better results (by discovering the optimal each time) with a 6 times speedup in simulation time.

3. Non-intrusive one-shot cache tuning in custom hardware

Section 2.2 described a method for tuning a highly configurable cache in a single pass. In this section, we will discuss modifications that convert SPCE into the one-shot hardware implementation.

3.1 Evaluation methodology

To determine the effectiveness of our hardware and ensure no degradation in the quality of results, we developed an evaluation methodology. To compare our technique with a state-of-the-art intrusive cache tuning heuristic, we implemented SPCE as an instruction trace processor in C++ for the configurable cache described by Zhang [28]. Using Simplescalar [7], we collected instruction trace files for a large selection of embedded system benchmarks from the Powerstone [15] and MediaBench [14] benchmark suites.

For comparison purposes, we first ran SPCE to determine the energy consumption of each cache configuration for each

```
process (addr )
  addr=addr>>w                                   //shift out word offset
  for b=bmax downto bmin                         // for each line size
    base_addr=addr>>$\log_2(b)$                  // shift out block offset
    was_found = lookup_stack ( base_addr ) // scan stack
    if ( was_found )
      for s = smin to smax                       // for each set
        num_conflicts =  count_set_conflicts ( s, base_addr ) // scan stack
        if ( num_conflicts <= amax )
          $\alpha$ = roundup(num_conflicts)      // next power of 2 assoc
          update_table ($\alpha$, s, b)          // mark appropriate table
  update_stack ( was_found, addr )               // push or move addr to top
```

Figure 1: Single-pass multi-cache evaluation algorithm.

benchmark using the same energy model described by Zhang [28]. We refer to these energy consumption values as the *gold* energy consumption of a configuration. We refer to the lowest energy consuming cache, or optimal cache configuration, as the G-Optimal (gold-optimal) cache. To determine energy savings compared to no cache configuration, Zhang also defines a *base cache* configuration consisting of an 8 KByte 4-way cache with a 32 byte line size. This configuration represents a commonly available configuration on a microprocessor and reflects the needs of the benchmarks studied.

Next, we modified SPCE into the one-shot tuner by implementing the changes discussed in sections 3.2 and 3.3. We ran each benchmark with the one-shot tuner to gather miss rates for each cache configuration and applied the same energy model to those miss rates to determine the optimal cache configuration – or the P-Optimal (predicted-optimal) cache. To determine the effectiveness of our predicted cache, we looked up the gold energy consumption for the P-Optimal cache and compared it to the gold energy consumption of the G-Optimal cache.

To predict address processing time in hardware, we augmented the software to count cycles expended per instruction and verified those counts with VHDL implementation.

3.2 Determining an upper bound on address processing time

Assuming a dual-ported SRAM for the stack, a hardware implementation of SPCE requires an average of 500 cycles of processing time per address, with a worst case processing time of nearly 7000 cycles for one address. Unfortunately, SPCE has only 1.5 cycles of available time on average to update all table locations, assuming an average CPI of 1.5. Given the complexity of the design and propagation delay, it is unlikely to be able to design hardware that could process each address in 1 to 2 reasonable length clock cycles.

As opposed to processing each address, the one-shot tuner samples the address stream. Address sampling means that the total number of hits will be incorrect, but if a good sampling rate is chosen, the *relative* hit rates will be correct, and the one-shot tuner will still determine the optimal cache configuration. We determined that a good sampling rate would not simply sample one instruction and then skip *m* instructions, but would instead buffer *n* instructions and then skip *m* instructions. We examined buffers sizes ranging from 2 to 64 and skip amounts ranging from 128 to 16384.

We evaluated each benchmark for each sampling rate and observed that different sampling rates were best for each benchmark. However, not wanting to specialize the sampling rate to each application, we chose buffer/skip rates of 4/128, 16/512, and 64/2048, sampling only 3% of the instruction stream, but providing cache tuning results within 4% of the optimal on average. Lower sampling rates resulted in configurations much further from the optimal. Among the three 3% rates, we chose 64/2048 as the best sampling approach, to allow the most absolute time to process the buffered addresses given that each address takes a variable amount of time to process. Assuming a CPI of 1.5, the rate translates to (2048*1.5) = 3072 cycles to process 64 addresses, meaning that the one-shot tuner has an upper bound of *48 cycles on average to process each instruction*.

Figure 2 compares the energy consumption normalized to the base cache configuration for a sampling rate of 64/2048 compared to the optimal cache configuration. On average, the sampling technique finds a cache configuration that is only 4% from the optimal. However, a few benchmarks resulted in suboptimal configurations. We note that these benchmarks performed poorly for all sampling rates studied. We examined the locality and execution breakdown of those benchmarks and did not discover any similarities and suspect that the regularity of the sampling rate is to blame for poor performance. We intend to study variable sampling rates in future work.

3.3 Custom hardware

SPCE's bottleneck is the stack scanning time. Each stack location is examined sequentially and each address can require up to 15 scans of the stack. If each search of the stack only accessed 3.2 locations on average, we could meet our deadline (3.2*15 = 48 cycles). We evaluated the benchmarks and determined that the average stack scan depth was 30 addresses. Clearly, a sequential scan of the stack was not going to be feasible and a parallel search of the stack was necessary

CAMs (Content Addressable Memory) are memory structures that allow for a single-cycle fully-associative search of the memory for a particular data item. TCAMs (Ternary Content Addressable Memory) have the same basic functionality as a CAM but allow for bits within the search data to be specified as don't care values (-). Thus, searching for the data value 0101- would produce a hit on both 01010 and 01011. TCAMs are heavily used in routing tables for network traffic [20].

A TCAM can readily be integrated into our hardware approach, eliminating the sequential search of the stack. For the initial stack search to determine a hit, the one-shot hardware would simply perform a lookup for the address with the block offset masked as don't care values. The TCAM will either flag the location of the data or indicate that the data doesn't exist. If more than one entry in the TCAM matches, the one-shot hardware is only interested in the highest priority match, or the value with the highest position in the stack and is specified by the TCAM.

If the address hits in the TCAM, the one-shot tuner determines, for each cache size, the minimum associativity necessary to yield a hit. Again, this can be done in a single cycle with a slight customization to the standard TCAM. The one-shot hardware searches the TCAM for matches of data items that map to the same set as the current address using don't care values. The TCAM outputs the number of matches that occurred, specifying the minimum associativity necessary to yield a cache hit.

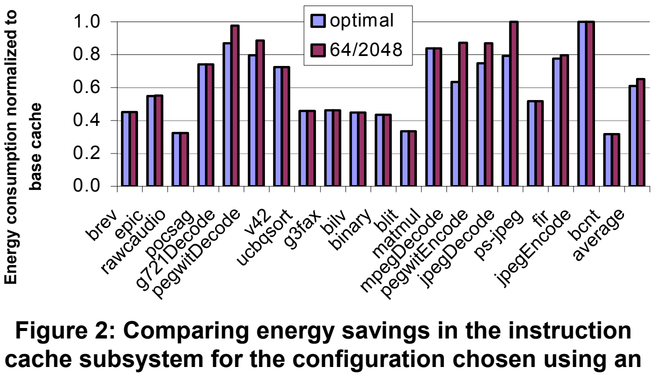


Figure 2: Comparing energy savings in the instruction cache subsystem for the configuration chosen using an address sampling technique compared to the optimal cache configuration.

After processing an address that yields a hit at the minimum block size, SPCE moves that address to the top of the stack. With a standard TCAM, such movement would require reading out and writing back every value to be moved. However, increasingly-common scan chains, if included in the TCAM and appropriately modified, allow all values in the TCAM to be shifted by one location in a single cycle. We include this functionality in our TCAM to improve stack updating time.

To get the fastest stack searching time, the one-shot hardware must read from the TCAM every cycle instead of wasting 1 cycle setting up each TCAM read. To eliminate the wasted time, the one-shot hardware utilizes speculative searches. While waiting for a read to be processed, the one-shot hardware sets up the inputs for the next read so that if the current read does not result in a hit, the next read will be processed on the following cycle. If there is a hit, the one-shot hardware simply discards the speculative values. This design sacrifices power in order to meet our timing deadline.

We evaluated different TCAM sizes ranging from 2 through 1024 entries. The remainder of the stack was stored in a dual-port SRAM and required sequential scanning and also used speculative reading. Using TCAM sizes of 256, 512, and 1024 we achieved average address processing times of 89, 61, and 50 cycles. Whereas a 1024-entry TCAM coupled with SRAM for the remainder of the stack nearly meets our timing constraints, the area overhead is too large.

To reduce the memory requirements, we studied the locality characteristics of the code. The premise is that, given the 90-10 rule [24], the most frequently executed instructions will remain near the top of the stack. If we restrict the stack size and the stack becomes full, the stack will drop the last address to make room for the new address.

We re-evaluated the benchmarks using TCAM sizes ranging from 2 to 1024 and stack size limits ranging from 8 to 1024. We discovered that by limiting the stack size to 512 entries, we could achieve identical energy savings. In addition, by implementing the entire stack in a TCAM, we could achieve an average address processing time of 47.9 instructions with individual benchmark values ranging from 40.2 to 51.

We implemented the custom one-shot hardware in synthesizable VHDL and verified cycle counts. We discovered that the maximum operating frequency of the design implemented in 0.18-micron technology was 200MHz. The update table operation was on the critical path and required determination of the correct layer and register values to update as well as incrementing a counter. We observed that the table update operation could be done in parallel and did not need to be on the critical path. To address this, we designed a custom co-processing circuit to encapsulate table updates.

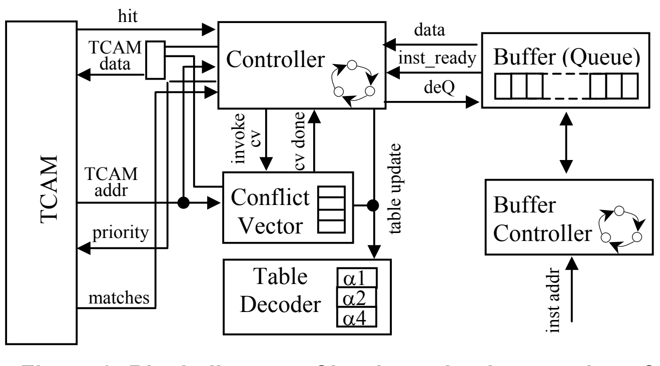


Figure 3: Block diagram of hardware implementation of the one-shot tuner. Signal detail has been simplified and control lines are omitted.

Figure 3 shows a block diagram of the major components of the one-shot tuner. The buffer controller is a simple controller responsible for applying the sample rate and filling the address buffer/queue. The queue is a simple producer/consumer buffer that buffers addresses to be processed and asserts a signal when an address is ready. For quickest processing, the controller can consume addresses and the buffer can fill the queue simultaneously. However, to eliminate the need for a dual-port memory, the output to the controller is stored in a separate register than can be read while the queue, implemented in a single-port SRAM, can be written to simultaneously. The conflict vector is responsible for handling details pertaining to minimum associativity requirements to yield cache hits. Figure 4 shows a simplified finite state machine for the controller. Only critical signals are shown and if not specified, a signal is implicitly set to 0.

4. Design evaluation

4.1 Experimental setup

We compare our non-intrusive one-shot tuner to a state-of-the-art intrusive heuristic (IH) in a runtime environment using the same configurable cache architecture as the evaluation methodology discussed in section 3.1 and detailed in [28].

For the one-shot tuner, we modeled the design in synthesizable VHDL and synthesized using Synopsys Design Compiler [25] to determine area requirements. We simulated the design at the gate level to capture switching activity and used Synopsys Power Compiler [25] to calculate power consumption. For TCAM power and energy consumption we utilized a TCAM power estimator [1]. We used the Artisan memory compiler [4] to model the queue

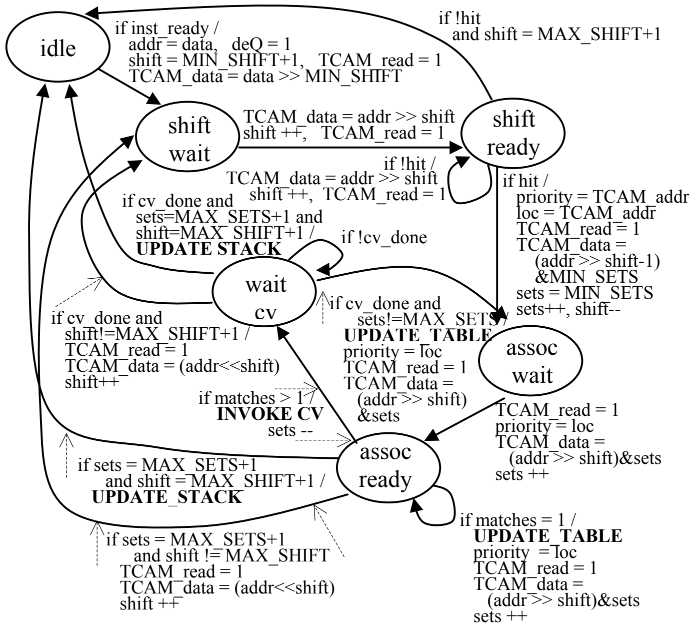

idle
shift wait
shift ready
wait cv
assoc wait
assoc ready
if inst_ready / addr = data, deQ = 1 shift = MIN_SHIFT+1, TCAM_read = 1 TCAM_data = data >> MIN_SHIFT
if !hit and shift = MAX_SHIFT+1
TCAM_data = addr >> shift shift ++, TCAM_read = 1
if !hit / TCAM_data = addr >> shift shift ++, TCAM_read = 1
if hit / priority = TCAM_addr loc = TCAM_addr TCAM_read = 1 TCAM_data = (addr >> shift-1) &MIN_SETS sets = MIN_SETS sets++, shift--
if cv_done and sets=MAX_SETS+1 and shift=MAX_SHIFT+1 / UPDATE STACK
if !cv_done
if cv_done and shift!=MAX_SHIFT+1 / TCAM_read = 1 TCAM_data = (addr<<shift) shift++
if cv_done and sets!=MAX_SETS / UPDATE_TABLE priority = loc TCAM_read = 1 TCAM_data = (addr >> shift) &sets
if matches > 1 / INVOKE CV sets --
TCAM_read = 1 priority = loc TCAM_data = (addr >> shift)&sets sets ++
if sets = MAX_SETS+1 and shift = MAX_SHIFT+1 / UPDATE_STACK
if sets = MAX_SETS+1 and shift != MAX_SHIFT TCAM_read = 1 TCAM_data = (addr<<shift) shift ++
if matches = 1 / UPDATE_TABLE priority = loc TCAM_read = 1 TCAM_data = (addr >> shift)&sets sets ++

Figure 4: State diagram for the one-shot tuner controller. If not specified, signals are set to 0.

as a single-port SRAM. We use 0.18-micron technology running at 1.8V for all components.

4.2 Energy and performance

We compare energy expenditures and tuning performance of the IH and our non-invasive one-shot tuner. The one-shot tuner runs the application for a single iteration with the base cache configuration while simultaneously gathering statistics for all cache configurations. The energy expended during tuning (tuning energy) is the energy consumed by the custom hardware. The one-shot method has no performance overhead. The IH explores many different configurations requiring the application to run for one iteration for each configuration explored. The tuning energy is the cumulative energy consumed by each configuration explored minus the energy expended if the base cache had run for those iterations. The performance overhead is the cumulative execution time required to explore each configuration minus the execution time of an equal number of application iterations running with the base cache configuration.

Figure 5(a) shows improvement in tuning energy by running the non-invasive one-shot tuner as opposed to the IH. On average, the one-shot tuner expends 4.6 times less tuning energy than the IH with energy savings for all except three benchmarks (values less than 1 result in more tuning energy by the one-shot tuner) and two of the benchmarks expend less than 2% additional energy. For the *blit* benchmark, the one-shot tuner expends 23% more tuning energy than the IH because the best configuration is the smallest cache configuration, which is determined after only 3 iterations of the IH.

Figure 5(b) shows speedup in tuning time for the non-intrusive one-shot tuner compared to the IH with an average speedup of 7.7 for all benchmarks.

4.3 Power and area overhead

The one-shot tuner shows both energy and performance improvements over the IH, but trades off increased area and a temporary increase in power consumption. In this section we compare these overheads to the ARM920T [3]. We point out that the reported area and power consumption for the ARM920T is likely optimistic while our numbers are pessimistic and actual overheads are likely to be less than what is reported here. The ARM920T consumes 160mW of power including the cache while running at 200Mhz. The reported area is 11.2 mm^2 including the cache.

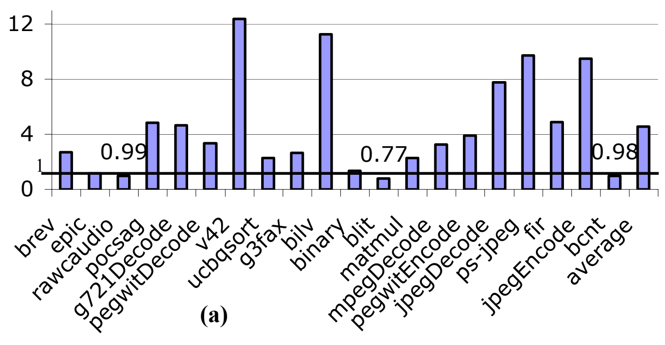
The custom hardware for our design excluding the TCAM and queue area is .25 mm^2. The area of the queue is .1 mm^2. Using a standard 16-transistor TCAM [13], we estimate the area as 16/6 = 2.7 times larger than an equivalent sized 6-transistor SRAM resulting in an area of .8 mm^2. We pessimistically estimate that our additional scan-chain logic may increase the TCAM area by as much as 25%, resulting in a total TCAM area overhead of 1 mm^2. The total area overhead of our one-shot tuner is 1.35 mm^2 - a 12% area overhead compared to the ARM920T.

The power consumption of our design excluding TCAM and queue access is 14.5 mW. One TCAM access consumes 177 mW of power [1] and read and write power of the queue is 45 mW and 49.5 mW respectively. TCAM power contributes heavily to overall power consumption because it is accessed nearly every cycle due to speculative searching. Queue accesses contribute very little to average power consumption because the queue is only accessed 6% of the time – 3% writes and 3% reads. The average power consumption of the one-shot tuner is 194 mW – a 2.2 times increase in power over the ARM920T. However, we point out that whereas this is a large increase in power, it is only the temporary active power during the short tuning cycle, and after tuning, the hardware would be shutdown and reactivated only during future tuning cycles. After tuning, the application need only iterate 4 times to recoup the expended power. For the ARM920T, the cache subsystem consumes nearly 50% of the total system power [21] and cache tuning can reduce power consumption of the cache subsystem by more than 50% resulting in a total system power reduction of approximately 25%.

5. One-shot tuner implementation methodology

For practical implementation, we propose replacing the IH with a one-shot cache tuner in certain operating environments. We examine both the IH and the one-shot tuner in a diverse environment such as a PDA intended to run many applications or a phase-based tuning environment where the cache would change during the run of an application to match different operating requirements for different phases of execution [23]. In both of these situations, the environment may change very quickly, only iterating a particular instance of an application or phase a few times before moving to a different instance. If the environment changes too quickly, the IH may not even have time to complete the tuning iterations before the environment changes.

We analyze the time needed to amortize tuning energy for both the IH and the non-invasive one-shot tuner targeted at the persistency of an application running on a PDA. The persistency



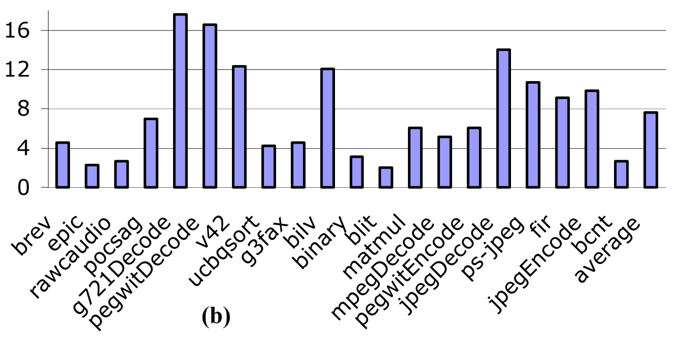


Figure 5: Improvements by a single-pass cache evaluation methodology versus a heuristic-based cache exploration methodology. (a) Improvement in energy expended (values less than 1 denote an increase in energy). (b) Tuning speedup obtained.

is measured as the number of iterations of an application before a new application is loaded. Over a period of time, a PDA user may start many applications but only use each of those for a short period of time.

Figure 6 evaluates different application persistency values. The x-axis shows the number of application iterations and the y-axis shows the average energy consumption across all benchmarks normalized to each application running with the base cache configuration. In each pair of bars, the first bar is the energy consumption of the IH and the second bar is the non-intrusive one-shot tuner. Each bar gives energy consumption broken into tuning energy and energy expended due to normal execution. A normalized value of 1 is the point at which the application has iterated long enough with the tuned cache to recoup the tuning energy. After only 2 iterations, the one-shot tuner has recouped all of the extra tuning energy while the IH takes 7 iterations to break-even. We also notice the time it takes for each method to converge on the maximum potential energy savings. The one-shot method converges to the maximum energy savings after 25 iterations and is within 5% of the maximum energy savings after only 10 iterations. The IH takes 75 iterations to converge on the maximum energy savings and comes within 5% after 50 iterations. Additionally, the tuning energy for the IH is not completely amortized until after 50 iterations.

6. Conclusions and future work

In this paper, we present the first run-time non-intrusive one-shot cache tuner for hardware implementation through modification of a simulation-based software methodology. We compare our one-shot tuner with an intrusive heuristic tuning method and show 4.6 times less tuning energy and a 7.7 times speedup in tuning time with acceptable size and power overhead. We look at a practical implementation of the one-shot tuner and in summary, we propose that the invasive heuristic is still the best option in situations with very high persistency. However, the non-invasive one-shot tuner is significantly better where applications or phases change regularly. Future work includes extending the one-shot tuner to evaluate multiple levels of cache.

7. Acknowledgements

This work was supported in part by the National Science Foundation (CNS-0614957) and the Semiconductor Research Corporation (2005-HJ-1331).

8. References

[1] B. Agrawal, T. Sherwood. Modeling TCAM power for next generation network devices. IEEE International Symposium on Performance Analysis of Systems and Software, 2006.
[2] D. Albonesi. Selective cache ways: on-demand cache resource allocation. MICRO 1999
[3] ARM, www.arm.com.
[4] Artisan. www.artisan.com
[5] R. Balasubramonian, D. Albonesi, A. Byuktosunoglu, S. Dwarkada. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. MICRO 2000.
[6] E. Berg, E. Hagerstein. StatCache: a probabilistic approach to efficient and accurate data locality analysis. IEEE International Symposium on Performance Analysis of Systems and Software, 2004.
[7] D. Burger, T. Austin, S. Bennet. Evaluating future microprocessors: the simplescalar toolset.University of Wisconsin-Madison. Computer Science Department Technical Report CS-TR-1308, July 2000
[8] A. Ghosh, T. Givargis. Cache optimization for embedded processor cores: an analytical approach. International Conference on Computer Aided Design, November 2003.
[9] T. Givargis. F. Vahid. Platune: a tuning framework for system-on-a-chip platforms. IEEE Transactions on Computer Aided Design, November 2002.
[10] A. Gordon-Ross, F. Vahid, N. Dutt. Automatic tuning of two-level caches to embedded applications. Design Automation and Test in Europe, Feb 2004.
[11] A. Gordon-Ross, F. Vahid, N. Dutt. Fast configurable-cache tuning with a unified second level cache. International Symposium on Low Power Electronics and Design, 2005.
[12] M. Hill, A. Smith. Evaluating associativity in CPU caches, IEEE Transactions on Computing, 1989.
[13] R. Kempke. A. McAuley. Ternary CAM memory architecture and methodology. U.S. Patent 5 841 874, Aug 13 1996
[14] C. Lee, M. Potkonjak, W.H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communication systems. MICRO 1997.
[15] A. Malik, W. Moyer, D. Cermak. A low power unified cache architecture providing power and performance flexibility. International Symposium on Low Power Electronics and Design, 2000
[16] R. Mattson, J. Gecsei, D. Slutz, I. Traiger. Evaluation techniques for storage hierarchies. IBM Systems Journal, 1970
[17] MicroBlaze, www.xilinx.com
[18] M. Palesi, T. Givargis, Multi-objective design space exploration using genetic algorithms. International Workshop on Hardware/Software Codesign, May 2002
[19] J. Pierper, A. Mellan, J. Paul, D. Thomas, F. Karim. High level cache simulation for heterogeneous multiprocessors, Design Automation Conference, 2004
[20] V. Ravikumar, R. Mahapatra, L. Bhuyan. EaseCAM: an energy and storage efficient TCAM-based router architecture for IP lookup. IEEE Transactions on Computers, May 2005.
[21] S. Segars. Low power design techniques for microprocessors, International Solid State Circuit Conf, 2001
[22] R. Sugumar, S. Abraham. Efficient simulation of multiple cache configurations using binomial trees. Technical Report CSE-TR-111-91, 1991.
[23] T. Sherwood, S. Sair, B. Calder. Phase tracking and prediction. 30th International Symposium on Computer Architecture, 2003
[24] D. Suresh, W. Najjar, F. Vahid, J. Villarreal, G. Stitt. Profiling tools for hardware/software partitioning of embedded aplications. LCTES 2003.
[25] Synopsys, www.synopsys.com
[26] Tensilica, Xtensa Processor Generator, www.tensilica.com/.
[27] P. Viana. A methodology to explore the design space of memory hierarchies for embedded systems. PhD Thesis, 2006
[28] C. Zhang, F. Vahid, W. Najjar. A highly-configurable cache architecture for embedded systems. 30th Annual International Symposium on Computer Architecture, June 2003.

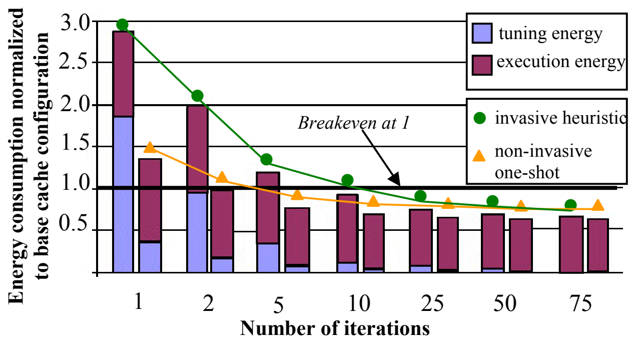


Figure 6: Average energy consumption across all benchmarks normalized to base cache configuration for different application persistency.