Configurable Cache Subsetting for Fast Cache Tuning*

Pablo Viana*, Ann Gordon-Ross†, Eamonn Keogh†, Edna Barros*, Frank Vahid†

*Centro de Informática
Federal University of Pernambuco
Recife-PE, Brazil

†Department of Computer Science and Engineering
University of California, Riverside
Riverside-CA, USA

ABSTRACT

Numerous variations of configurable caches, having variable parameters like total size, line size, and associativity, have been proposed or have appeared in commercial microprocessors in recent years. Tuning a configurable cache to a target application has been shown to reduce memory-access power by over 50%. However, searching the configuration space for the best configuration can require much time or power, even when using recent cache tuning heuristics. We sought to determine, for a particular domain of applications, the smallest subset of cache configurations that would still enable effective tuning. For a suite of 34 benchmarks and a cache with 18 possible configurations, we determine through an exhaustive search of all possible subsets, that only 3 or 4 candidate configurations are necessary to support tuning. We introduce a new heuristic, adapted from an efficient and effective heuristic developed for data mining, to quickly determine the best configurations for any sized subset, with near optimal results. We then consider a configurable cache with 17,640 possible configurations and improve our heuristic to include a pre-pruning step, yielding near optimal tuning results. We conclude that only 3 or 4 possible cache configurations are needed to offer a near optimal configuration for every benchmark in our suite - resulting in a 91% reduction in design space exploration time over a state-of-the-art cache tuning heuristic.

Categories and Subject Descriptors

B.3 [**Memory Structures**]: Performance Analysis and Design Aids

General Terms

Algorithms

Keywords

Configurable cache tuning, cache optimization, low energy.

*Supported by CAPES Foundation process number BEX1366/04-1

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

1. INTRODUCTION AND MOTIVATION

Every application has different cache requirements with respect to total size, line size, and associativity [16]. However, platforms tend to offer a fixed memory hierarchy configuration that works well for all applications but is rarely the lowest energy or highest performing cache for any single application. For example, while a 32-byte line size is quite common, studies have shown that many applications benefit from a 16-byte or 64-byte line size [17]. An application with low spatial locality and a large line size would expend many cycles fetching data that is never referenced. On the other hand, an application with high spatial locality and a small line size would expend many cycles fetching subsequent data that could have been prefetched with a larger line size. Using a 32-byte line size offers a trade-off between both extremes.

Similarly, the cache size should reflect the size of the application's working set. An undersized cache would cause trashing with the working set constantly being swapped in and out of the cache. An oversized cache would expend excessive energy fetching from the larger cache. Similarly, the associativity should reflect the needs of the application. By tuning these parameters, the energy consumed by the memory hierarchy can be reduced by more than 50% and execution time can be reduced by 30% [4, 8, 13, 17].

To facilitate cache tuning, configurable caches are available in both soft-core [3, 1, 2, 15] and hard-core [4, 13, 16] processors. In a soft-core processor, a designer sets the cache's parameters in a synthesis tool, generating a customized cache. In a hard-core processor, a configurable cache appears on a physical device. Software-configurable registers control muxes and the cache controller so that cache parameters can be varied.

For soft- or hard-core configurable caches, a designer is likely responsible for determining appropriate cache parameters, however this process may be quite difficult. One method involves simulating all possible cache configurations. Simulating a real application (not just small benchmark kernels) for one cache configuration, even using a fast instruction-set simulator, may require dozens of minutes or even hours, resulting in many hours, days, or perhaps weeks of simulation time. Additionally, simulation time can be greatly increased when considering compounding factors such as simulating an application running under an operating system, or simultaneously considering other configurable architecture features (configurable microprocessor datapaths, configurable buses, etc.) that multiplicatively enlarge the total architecture configuration space. To reduce simulation time, cache tuning heuristics have been proposed [5, 8, 16] to speedup cache exploration time and can search the configuration space up to 500x faster, however even with a state-of-the-art heuristic, dozens of cache configurations may need to be simulated.

To eliminate costly simulation time, a hard-core processor can tune the cache dynamically. Cache tuning heuristics can be implemented in specialized hardware to perform feedback directed cache tuning either at system startup or dynamically during runtime [16]. However, each cache configuration explored introduces power and performance overhead.

Both the simulation-based and dynamic tuning methods can benefit greatly by any reduction in the number of cache configurations explored. Even simulating 7-10 configurations can require many hours or days. Furthermore, two level configurable cache hierarchies may themselves possess tens of thousands of possible configurations, such that even fast heuristics may still explore dozens of cache configurations, and may introduce too much power and performance overhead in a dynamic tuning environment.

To further reduce the number of configurations that must be examined, we consider the situation where a microprocessor vendor intends to support applications similar to one or more found in a particular benchmark suite, such as media processing benchmarks, networking benchmarks, or control system benchmarks. Of course, a vendor can support applications from many different benchmark suites, asking the user to identify the most appropriate suite for a target application. For a particular benchmark suite, we address the question:

"Can we select a very small subset of possible cache configurations, perhaps just 2 to 4 configurations, so that a tuning tool can be restricted to searching only that small subset and achieve power and execution time reductions close to those obtained when searching the complete cache configuration space (either exhaustively or using a proven search heuristic)?"

A configurable microprocessor vendor would determine that small cache subset for a benchmark suite, and could develop cache tuning tools that only search that subset, thus improving tuning tool runtime in a soft-core simulation approach, and tuning tool power and execution time overhead in a hard-core dynamic approach.

In Section 2 of this paper, we provide results of an exhaustive cache configuration subset search using a single level of cache to find the optimal subset. In Section 3, we introduce a fast yet effective $O(n^2)$ heuristic that determines near-optimal subsets using a heuristic adapted from a sophisticated data clustering heuristic (Keogh's heuristic) originally developed for segmenting time series. In Section 4, we extend Keogh's heuristic with a pre-pruning stage to consider a two level cache hierarchy with $17,640$ possible configurations. In Section 5, we show that restricting a tuning tool to a pre-determined configuration subset results in faster tuning with very little error, compared to tuning method that considers all configurations, whether searching the space heuristically or exhaustively.

2. OPTIMAL CACHE CONFIGURATION SUBSET FOR A ONE-LEVEL CACHE

2.1 Problem Definition

Consider a set of n applications $A = \{a_1, a_2, a_3, ..., a_n\}$ intended to run on a configurable cache architecture capable of supporting the design space $C = \{c_1, c_2, c_3, ..., c_m\}$ of m possible cache configurations. We define $e(c_j, a_i)$ as the total energy consumed by running application a_i on the architecture with cache configuration c_j. We also define $c_o \in C$ as the optimal cache configuration for the specific application a_i, such that $e(c_o, a_i) \leq e(c_j, a_i), \forall c_j \in C$.

The problem is to determine the p configurations that compose the subset $C^{'} \subset C$, capable of offering a configuration $c_o^{'} \in C^{'}$ with energy savings near to the optimal configuration for each particular application, $c_o \in C$. Through exhaustive exploration of all possible combinations of extracted subsets $C^{'} = \{c_1, c_2, ..., c_p\}$ from $C = \{c_1, c_2, c_3, ..., c_m\}$, we can select the best combination based on the lowest average energy increase $e_{inc}(c_o^{'}, a_i)$ by using $c_o^{'}$ instead of c_o (Equation 1).

$$e_{inc}(c_o^{'}, a_i) = \frac{e(c_o^{'}, a_i) - e(c_o, a_i)}{e(c_o, a_i)} \quad (1)$$

In other words, for a given subset of configurations, we compute for each application the energy increase that results when restricted to choosing the best configuration from that subset, compared to choosing the best configuration from the full set of cache configurations. We then compute the average energy increase across all the applications.

The number of combinations N for selecting a subset with p configurations from a set with m configurations can be calculated by the combinatorial of m and p (Equation 2).

$$N(p) = \frac{m!}{p!(m-p)!} \quad (2)$$

2.2 One-Level Cache Architecture

For the single level cache hierarchy, we explore separate level one instruction and data caches. Each cache utilizes the configurability presented by Zhang et. al [17]. The cache is composed of four configurable banks each of which acts as a way in the cache - thus the base cache is a 4-way set associative cache. The ways/banks may be selectively shut down or enabled to offer configurable size. Additionally, ways may be concatenated to offer configurable associativity. For example, given a base cache of size 8 kByte composed of four 2 kByte banks, way shutdown offers 2 kByte and 4 kByte cache size variations. Way concatenation allows ways to be logically concatenated to facilitate configurable associativity offering direct-mapped, 2-way, and 4-way set associativity configurations for a cache composed of 4 separate banks. Due to the specifics of the cache layout, 2 kByte 2- and 4-way set associativities and 4 kByte 4-way set associativity are not available. The cache offers a base physical line size of 16 bytes with configurability to 32 and 64 bytes by fetching/probing subsequent lines in the cache. Zhang et. al [17] presents hardware layout verification for their configurable cache and shows that the configurability does not impact access time.

2.3 Experimental Setup

We determine the optimal cache configuration subsets for the set A of $n = 34$ different applications from the MediaBench [12], Powerstone [13], and EEMBC [7] benchmark suites. The configurable cache offers the $m = 18$ distinct configurations shown in Table 1, where each configuration is designated with a value c_j.

	16B	32B	64B
2k_1W	c_1	c_7	c_{13}
4k_1W	c_2	c_8	c_{14}
4k_2W	c_3	c_9	c_{15}
8k_1W	c_4	c_{10}	c_{16}
8k_2W	c_5	c_{11}	c_{17}
8k_4W	c_6	c_{12}	c_{18}

Table 1: One-level cache configurations

For example, a 4 kByte direct-mapped cache with a 32 byte line

size is designated as c_8. These designations will be used throughout the remainder of this paper to identify each particular cache configuration. For comparison purposes, we choose the cache configuration c_{18} as our base cache configuration [17].

We determine the total energy $E(total)$ consumption for a particular cache configuration using the following equations combining dynamic (dyn) and static (sta) energy of the caches, CPU stall energy, and off chip access energy:

$$E(total) = E(sta) + E(dyn)$$
$$E(dyn) = cache\_hits \times E(hit) + cache\_misses \times E(miss)$$
$$E(miss) = E(offchip\_access) + miss\_cycles \times E(CPU\_stall) + E(cache\_fill)$$
$$miss_cycles = cache\_misses \times miss\_latency + (cache\_misses \times (linesize/16)) \times memory_bandwidth)$$
$$E(sta) = total\_cycles \times E(static\_per\_cycle$$
$$E(static\_per\_cycle)) = E(per\_kByte) \times cache\_size\_in\_kbytes$$
$$E(per\_kByte) = \frac{E(dyn\_of\_base\_cache) \times 10\%}{base\_cache_size\_in\_kBtes}$$

We obtain dynamic energy for each cache configuration using the Cacti model [14] for 0.18-micron technology. We estimate static energy as 10% of the dynamic energy - a reasonable assumption for present technology. We use SimpleScalar [6] to obtain cache hit and miss values for each cache configuration. We obtain off-chip access energy from a standard low-power Samsung memory. CPU stall energy is quite difficult to determine given its high dependency on the actual microprocessor utilized. We analyzed a variety of different microprocessors and determined that the stall energy was typically around 20% of the active energy. We use this estimated value so that the results presented in this paper are applicable to a wide variety of final system configurations. For miss penalties, we estimate that a fetch from off-chip memory will take 40 times longer than a fetch from the level one cache and we estimate the bandwidth as 50% of the miss penalty. Previous work shows the fidelity of cache tuning heuristics across different miss penalty and bandwidth values [8].

2.4 Results

We exhaustively evaluated all possibilities for choosing the cache configurations to compose the subset $C^{'}$ under all subset sizes ($1 \leq p \leq 18$). In other words, we evaluated $\sum_{p=1}^{18} N(p) = 262,143$ possible subsets. To accomplish this, we first generated complete data for every application running on every cache configuration. We then wrote a script that, for each subset size, generated all possible subset combinations. We determined the best possible subset of cache configurations $C^{'}$ from C for all subset sizes by computing the average increase in energy for choosing the lowest energy cache configuration $c_o^{'}$ from the subset $C^{'}$ as opposed to the optimal configuration c_o for each benchmark from the entire set C. Running the script to exhaustively search for the best subsets required 14 minutes on a 1 GHz Pentium 3 with 512 MB RAM.

The results are presented in Figure 1 and show that no energy penalty is perceived by reducing the design space to a subset of half of the total configuration space ($p = 9$). Results also show that subsetting has the ability to produce near optimal results (energy increase lower than 5%) as the subset size is decreased down to 4 for the data cache and down to 3 for the instruction cache.

The results observed using exhaustive search methods motivate the feasibility of tuning tool vendors creating tools that search just a few carefully selected configurations for a particular domain. We point out that an end-user (the tuning tool user) would not be expected to utilize such exhaustive search methods - the point is that, through extensive pre-investigation by the tool vendor, the tuning tool runtime can be accelerated for the end user, by simply searching approximately 4 cache configurations. As we planned to consider a two-level cache hierarchy with $17,640$ configurations, we knew that the exhaustive method of subset exploration would need to be improved: Our one-level cache with just 18 configurations required 14 minutes to search $262,143$ possible subsets. $17,640$ configurations would result in a combinatorial explosion of subsets that would not be computable in reasonable time. Thus, we sought to develop an effective heuristic for subsetting our one-level cache, with plans to then extend that heuristic to our two-level cache.

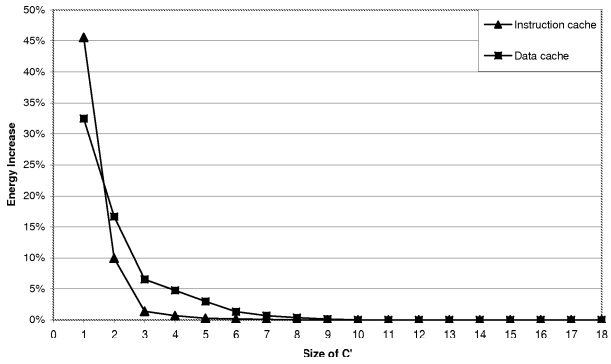

Figure 1: Average increase in energy over the optimal cache configuration by choosing the lowest energy cache from the best subset C' instead of from C

3. FAST ONE-LEVEL CONFIGURABLE CACHE SUBSETTING USING KEOGH'S HEURISTIC

We sought to develop a heuristic to reduce the time for subsetting $C^{'}$ from C, while maintaining near optimal results. We first tried a straightforward heuristic that, from the complete data set of every application running on every configuration, chose the p configurations that offered optimal energy for the largest number of applications.

However, this heuristic, while finding the best or near best configuration for many applications, performed poorly for certain applications. Selecting a subset $C^{'}$ with 4 cache configurations, the average increase in energy consumption was a very poor 44% for instruction cache and 15% for data cache - far worse than the optimal subsets.

We then considered a traditional hierarchical clustering heuristic. We defined the similarity between cache configurations in terms of average energy savings for all benchmarks to hierarchically group configurations and thus select representative configurations from each group, clustering until we had a number of groups equal to the desired subset size. Selecting a subset $C^{'}$ with 4 cache configurations, the average increase in energy compared to the exhaustive approach was about 1.3% for instruction cache and 16% for data cache. This was much better than the previous heuristic, but still had room for improvement.

The main problem with a traditional clustering approach is that the approach does not consider the overall impact on energy across the benchmarks when removing one given configuration c_j from C. After investigating clustering methods, we found a problem similar to our subsetting problem. This other problem has been studied extensively and has a sophisticated high-quality efficient heuristic solution.

3.1 Keogh's Clustering Heuristic

Our cache configuration subsetting problem is similar to the problem of segmenting time series. The segmenting time series problem seeks to find the best number of clusters, and the clustering itself, which best groups a series of data, subject to some criteria that defines error from the original unclustered data. Such a method is known fundamentally to data mining techniques. Actually, it builds upon related work in computer graphics known as decimation methods [10]. The clustering can be applied, for example, to reduce the number of primary colors for displaying an image by merging color nuances according to the associated error/difference in the final image. The idea is to iteratively eliminate colors and just leave the indispensable ones, those of which that can nearly represent the removed colors, reducing the data needed to store the image.

In our study, we adapted the recently introduced but now widely referenced and commonly utilized clustering heuristic proposed by Keogh [11], to eliminate cache configurations from an initial design space C by just leaving the p promising configurations, thus comprising the restricted space $C^{'}$ of subsets.

Keogh's heuristic, as applied to our problem, looks for adjacent cache configurations c_j and c_k and iteratively merges c_j into c_k or c_k into c_j, eliminating from the design space C the configuration (c_j or c_k) whose absence minimizes the average increase in the energy consumption over all benchmarks. Here, two cache configurations c_j and c_k are said to be adjacent in the current design space if their location in the design space differs by the fewest number of cache parameters as possible.

This process of elimination of configurations from the design space means that if c_k is eliminated from C by merging into c_j, the estimated energy $e_{inc}(c_k, a_i)$ to run a_i on c_k will be evaluated by using the energy to run a_i on c_j. The impact $i(c_j, c_k)$ for replacing c_k by c_j is evaluated by the average increase in energy of $c_o^{'}$ for all applications in A (Equation 3).

$$i(c_j, c_k) = \frac{1}{n} \cdot \sum_{i=1}^{n} e_{inc}(c_o^{'}, a_i) \quad (3)$$

where $c_o^{'} \in C^{'}$, which is the new design space iteratively obtained by removing c_k.

```
Input: Energy matrix estimate E_inc
Output: C'

Begin

  C' = C;
  While (C'>4)
     For all adjacent pair cj,ck
        i(cj,ck)=evaluate_merge(cj,ck);
     End
     [cj,ck] = minimum_pair(i);
     Merge(cj,ck);
     C' = C'- ck;     //remove configuration
  End
  Return (C');

End.
```

Figure 2: Proposed variation of Keogh's heuristic.

Figure 2 presents our adaptation of Keogh's heuristic to select cache configurations for the restricted design space. The complexity of the heuristic is $O(N^2)$, where N is the size m of the design space C.

3.2 Results

We apply Keogh's heuristic to the same set of 34 benchmarks and design space of 18 cache configurations as described earlier. Figure 3 compares the energy consumption (normalized to the base cache configuration c_{18}) for the optimal energy cache configuration in the entire design space C (all 18 configurations) and the lowest energy cache configuration available for each design space $C^{'}$ (4 configurations chosen by the heuristic in Figure 2) and $C^{'*}$ (4 configurations chosen by exhaustive simulation). Keogh's heuristic performs extremely well, by including a near optimal cache configuration in the restricted design space for nearly every application.

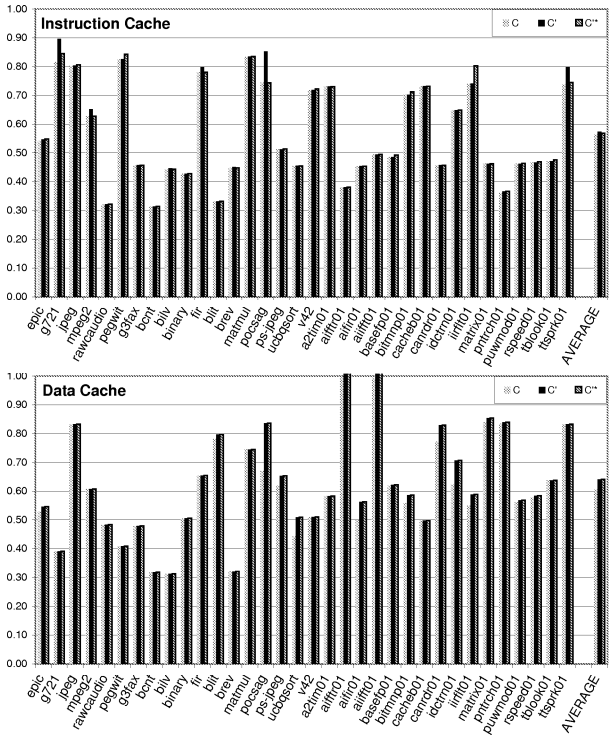

Figure 3: Energy consumption for a single level of cache normalized to the base cache configuration for the optimal cache configuration (C), the lowest energy cache configuration chosen from the subset determined either using the adaptation of Keogh's heuristic ($C^{'}$) or using the exhaustive subsetting $C^{'*}$, for the instruction cache and for the data cache.

Table 2 summarizes the accuracy of the results and selected configurations obtained by using the exhaustive approach and Keogh's heuristic, again running on a 1 GHz Pentium 3 processor with 512 MB RAM. Keogh's heuristic produces results very near those of exhaustive search, but with a speed up of 843.

Heuristic	Runtime (ms)	Average error		Subset $C^{'}$
Exhaustive	835,000	I$	0.67%	c_7, c_9, c_{10}, c_{14}
		D$	4.75%	c_1, c_3, c_5, c_{13}
Keogh's	991	I$	0.69%	c_7, c_8, c_9, c_{16}
		D$	4.75%	c_1, c_3, c_5, c_{13}

Table 2: Comparing the heuristics

4. FAST TWO-LEVEL CONFIGURABLE CACHE SUBSETTING USING FURTHER HEURISTIC EXTENSION

For the two-level cache, we include the first level configurable cache described in Section 2.2. However, for the second level of cache we explore a unified cache structure quite different from the traditional unified cache. The cache utilizes the way management methodology implemented in the M*CORE microprocessor [13]. Way management allows for each way to be specified to cache instructions only (I), data only (D), instructions and data (unified or U), or the way may be shutdown altogether (empty or E). For the level two cache, we use a base cache of 64 Kbytes composed of four configurable ways. The level two cache also offers a physical line size of 16 bytes with configurability to 32 and 64 bytes.

The multilevel cache hierarchy, being composed of separated level one caches and the unified second level featuring way management, offers $17,640$ possible configurations. Using the same energy equations as described in Section 2.3, we estimate that a fetch from the level two cache will take 4 times longer than a fetch from the level one cache and a fetch from off-chip memory will take 10 times longer than a fetch from the level two cache.

Utilizing the exhaustive subset searching of $17,640$ configurations would require impossibly long computation time. Furthermore, even applying Keogh's heuristic to the design space of $17,640$ configurations with no pruning would require evaluating more than three hundred million adjacent configurations. For a single level cache, just 306 evaluations of adjacent configurations were needed. This estimation is calculated by: $\sum_{p=2}^{m}(p-1)$, which is the number of adjacent pairs (c_j, c_k) in the subset C while the design space iteratively shrinks from m down to 2 configurations. Besides the huge amount of memory needed (over 300 MB), we estimate that the two-level cache would take about 280 hours to be processed by the Keogh's heuristic on the Pentium host computer used for the experiments.

We thus decided to prune cache configuration space, prior to applying Keogh's heuristic. We observed in our prior experiments that each configuration in the best subset was in fact the optimal configuration for a least one benchmark. Thus, we decided to use a pruning method that found the optimal cache configuration for every benchmark, and then took the union of those configurations as the set of possible configurations as input to our subsetting heuristic. Yet, finding the optimal configuration for a particular benchmark would require $17,640$ simulations for each benchmark, one simulation for each cache configuration. Thus, we use the heuristic described by Gordon-Ross [9] to find the best cache configuration for each application. Although such a heuristic carefully examines only 34 configurations from the design space of 17,640 configurations, it was shown to yield near-optimal results. However, running a cache configuration heuristic that explores only 34 configurations may still require several hours of simulation time, which is why we wish to determine if a tuning tool vendor can narrow the search to only 4 or so configurations, for very fast tuning by the end user.

4.1 Results

We considered 17 applications, which we have carefully verified the accuracy of the Gordon-Ross' heuristic to find the optimal cache. In some cases, two equally good (optimal) configurations were found for one application, resulting in 26 cache configurations that were each best for at least one application. We ran each of the 17 applications on each of the 26 cache configurations, and then applied Keogh's heuristic to find the best subsets from the 26 configurations.

	I$ Level 1	D$ Level 1	Cache Level 2
$c_1^{'}$	4kB 1W 16B	2kB 1W 16B	64kB 4W 16B DEII
$c_2^{'}$	4kB 1W 16B	8kB 2W 16B	64kB 4W 16B DEII
$c_3^{'}$	8kB 1W 16B	8kB 4W 16B	64kB 4W 16B DDEU
$c_4^{'}$	8kB 4W 16B	8kB 1W 16B	64kB 4W 16B DEEI

Table 3: Cache configuration subset chosen by Keogh's heuristic for the highly configurable two-level cache.

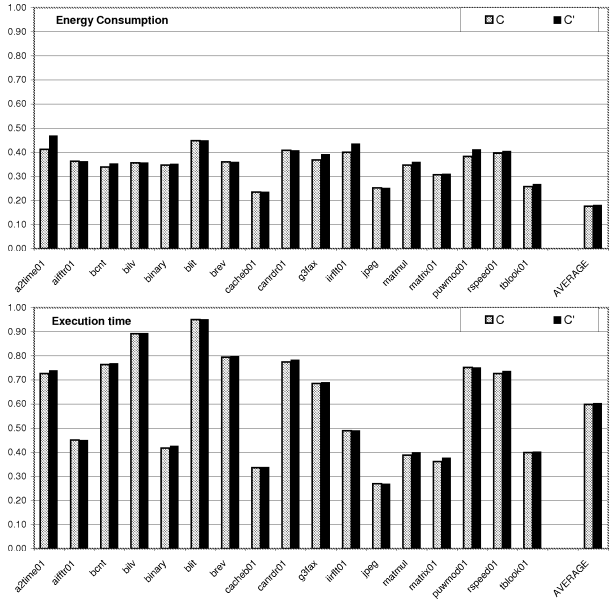

Figure 4: Energy consumption and Execution time of the two-level cache for the optimal configuration and the lowest energy configuration determined using Keogh's heuristic.

Table 3 presents the 4 configurations selected by Keogh's heuristic, where $c_1^{'}$ for example, denotes the cache configuration composed of a level one direct-map instruction cache (total size 4 kBytes) and data cache (total size 2 kBytes), both with 16 bytes per line. The level two cache has a total size of 64 kBytes, a line size of 16 bytes, and 1 data way (D), 2 instruction ways (I), and 1 way shutdown (E). Figure 4 compares the results obtained by using the best configuration $c_o^{'}$ from a subset of 4 configurations $C^{'}$ chosen by Keogh's heuristic and the optimal configuration $c_o \in C$ from the $17,640$ configurations obtained by Gordon-Ross's heuristic. The selected subset $C^{'}$ composed of just 4 configurations offers near-optimal results, with an average energy increase of just 3.36% over the optimal cache configuration from C. In addition, we sought to evaluate the impact on the execution time for every application when the design space is restricted to $C^{'}$ (4 configurations). Figure 4 shows also that the average increase in the execution time is just 0.55% compared to the optimal cache configuration c_o. Despite the pruning phase that reduced the design space to just 26 configurations, the use of exhaustive subsetting presented in Section 2 would take around 53 hours to be completed due to the exponential growth in runtime - if we consider 30 configurations it would require an estimated 36 days. Extrapolating the projection, we estimate that 35 configurations would take 3 years and 40 configurations would take more than 100 years to be completed. Keogh's heuristic, being $O(N^2)$, grows more reasonably, requiring just seconds for 26 configurations, and less than a minute even for 40 configurations.

5. CACHE CONFIGURATION SUBSETS FOR BENCHMARK SUITES

In order to observe the behavior for subsetting a configurable cache for a specific application domain, we applied Keogh's heuristic on the two-level highly configurable cache and selecting two different groups of applications. One group is composed of 8 automotive benchmarks from EEMBC suite: a2time01, aifftr01, cacheb01, canrdr01, matrix01, puwmod01, rspeed01, and tblook01. The other group is composed of 8 more diverse benchmarks from different domains from Powerstone and EEMBC: bcnt, bilv, binary, blit, brev, iirflt01, jpeg, and matmul. The results in Figure 5 shows that for the tightly constrained benchmark suite, the Automotive domain, just 2 cache configurations can still yield energy savings no worse than 5% compared to the optimal configuration among the whole design space. However, for the other group, at least 4 different cache configurations are necessary to ensure energy increase lower than 5%.

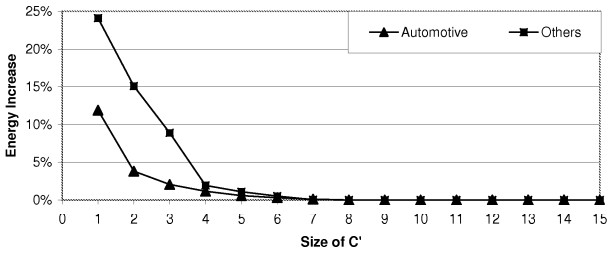

Figure 5: Average energy increase over the optimal cache configuration by choosing the lowest energy cache from the subset $C^{'}$ instead of from C.

6. CONCLUSIONS AND FUTURE WORK

This paper introduced the cache configuration subsetting problem. Through exhaustive configuration subset exploration of a one-level cache with 18 possible configurations, we found that we could select four cache configurations that would provide enough configurability to attain near-optimal energy savings for a set of 34 benchmarks. We showed that by adapting Keogh's heuristic, previously developed for segmenting time series, to the cache subsetting problem, we could find near optimal cache configuration subsets 800x faster than exhaustive subsettting. We extended the subsetting problem to a highly-configurable two-level cache with over $17,000$ possible configurations. We heuristically pruned the search space down to 26 configurations, and again show that Keogh's heuristic achieves the near-optimal subsetting of those 26 configurations. The conclusion is that, if a tuning tool vendor can identify a set of benchmarks that encapsulate a domain, then a tuning tool end user can benefit from searching only 3 or 4 configurations, rather than the 34 configurations explored using a cache exploration heuristic resulting in a 91% reduction in exploration time, reducing a several hour job to just a few dozen minutes, or a several day job to several hours. As platform tuning is becoming an increasingly important topic, especially in the presence of highly-configurable microprocessor platforms enabled by soft-core processors on FPGAs, and with additional configurable architecture parameters relating to the processor configuration, the tuning problem becomes increasingly important. The domain-specific subsetting approach may be useful not just in cache configuration, but in the broader space of processor architecture configuration tuning as well - and we expect the heuristic we applied will be useful in those situations. We plan to investigate broader configuration tuning in future work.

7. REFERENCES

[1] Arc international. In http://www.arccores.com, 2005.

[2] Arm embedded processor. In http://www.arm.com, 2005.

[3] Nios embedded processors. In http://www.altera.com, 2005.

[4] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. Journal of Instruction-Level Parallelism, 2, May 2000.

[5] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture, pages 245–257, New York, NY, USA, 2000. ACM Press.

[6] D. Burger, T. M. Austin, and S. Bennet. Evaluating future microprocessors: the simplescalar tool set. Technical Report CS-TR-1996-1308, Computer Sciences Department, University of Wisconsin, Madison, WI, August 1996.

[7] EEMBC. The Embedded Microprocessor Benchmark Consortium. In http://www.eembc.org, 2005.

[8] A. Gordon-Ross, F. Vahid, and N. Dutt. Automatic tuning of two-level caches to embedded applications. In DATE '04: Proceedings of the conference on Design, automation and test in Europe, February 2004.

[9] A. Gordon-Ross, F. Vahid, and N. Dutt. Fast configurable-cache tuning with a unified second-level cache. In ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design, pages 323–326, New York, NY, USA, 2005. ACM Press.

[10] P. S. Heckbert and M. Garland. Survey of polygonal surface simplification algorithms, multiresolution surface modeling course. In Proceedings of the 24th International Conference on Computer Graphics and Interactive Techiniques, 1997.

[11] E. J. Keogh, S. Chu, D. Hart, and M. J. Pazzani. An online algorithm for segmenting time series. In ICDM '01: Proceedings of the 2001 IEEE International Conference on Data Mining, pages 289–296, Washington, DC, USA, 2001. IEEE Computer Society.

[12] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In International Symposium on Microarchitecture, pages 330–335, 1997.

[13] A. Malik, B. Moyer, and D. Cermak. A low power unified cache architecture providing power and performance flexibility (poster session). In ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design, pages 241–243, New York, NY, USA, 2000. ACM Press.

[14] G. Reinman and N. Jouppi. Cacti 2.0: An integrated cache timing and power model. Technical report, COMPAQ Western Research Lab, 1999.

[15] Tensilica. Xtensa Processor Generator. In http://www.tensilica.com, 2005.

[16] C. Zhang, F. Vahid, and R. Lysecky. A self-tuning cache architecture for embedded systems. In Proc. of the Design, Automation and Test in Europe(DATE'04), February 2004.

[17] C. Zhang, F. Vahid, and W. Najjar. A highly configurable cache for low energy embedded systems. Trans. on Embedded Computing Sys., 4(2):363–387, 2005.