An Application Classification Guided Cache Tuning Heuristic for Multi-core Architectures

Marisha Rawlins and Ann Gordon-Ross*
Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611, USA
mrawlins@ufl.edu & ann@ece.ufl.edu
**Also with the NSF Center for High-Performance Reconfigurable Computing*

Abstract—Since multi-core architectures are becoming more popular, recent multi-core optimizations focus on energy consumption. We present a level one data cache tuning heuristic for a heterogeneous multi-core system, which classifies applications based on data sharing and cache behavior, and uses this classification to guide cache tuning and reduce the number of cores that need to be tuned. Results reveal average energy savings of 25% for 2-, 4-, 8-, and 16-core systems while searching only 1% of the design space.

I. Introduction and Motivation

Multi-core system optimizations improve system performance [1][3][10] and energy consumption [6][7][10] by tuning (specializing) the system to the application's runtime behavior and resource requirements. Many multi-core optimizations leverage single-core optimization fundamentals, however, multi-core optimization development introduces additional challenges with respect to disparate multi-core architectural layouts, application decomposition, and core interactions.

Heterogeneous architectures contain cores with different configurations and are more difficult to tune than homogeneous architectures with identical cores due to a much larger design space. However, this increased design space reveals the potential for higher energy savings when different cores/applications have different resource requirements. Additionally, a data-sharing application's behavior may change if a core's optimization affects the behavior of the applications executing on the other cores, leading to circular optimization dependencies between cores ([4] showed a similar circular dependencies between level one and level two caches). Finally, single-core optimizations did not have to consider shared resource contentions and core interactions, and in multi-core systems, isolating the cores' behaviors may not be possible.

In this paper, we focus on runtime cache tuning, a prominent single-core optimization due to the memory hierarchy's large impact on system performance and energy consumption [9]. Cache tuning determines the optimal, lowest energy cache configuration—specific values for cache parameters such as size, line size, and associativity—that matches an application's runtime behavior, and achieves energy savings as high as 62% in single-core systems [5]. Cache tuning requires a configurable cache architecture (e.g., [17]) with tunable cache parameters, whose values can be specified/changed during runtime.

A cache tuner explores the configurable cache design space using the following cache tuning process: 1) execute the application for one tuning interval in each potential configuration (tuning intervals must be long enough for the cache behavior to stabilize); 2) gather cache statistics, such as the number of accesses, misses, and write backs, for each explored configuration; 3) combine the cache statistics with an energy model to determine the optimal cache configuration; and 4) fix the cache parameter values to the optimal cache configuration's parameter values.

During design space exploration, cache tuning incurs energy and performance penalties while executing inferior, non-optimal configurations. Minimizing these cache tuning overheads is critical in a multi-core system due to overhead accumulation across each core and the potential power increase if all cores simultaneously tune the cache. Additionally, applications with core interactions have circular tuning dependencies where tuning one core's cache affects the behavior of the other cores' caches. For example, increasing the cache size increases the amount of data that the cache can store and decreases the miss rate. However, this larger cache is more likely to store shared data, which may increase the number of cache coherence evictions and forced write backs for all cores, thus increasing energy consumption.

The cache tuner's design space exploration method is critical to mitigating the cache tuning overhead. Since exhaustive design space exploration is infeasible during runtime, tuning heuristics quickly find the optimal or near optimal configuration. Single-core cache tuning heuristics can prune the design space to a fraction of the configurations (0.2%) and still determine configurations within 1% of the optimal [5]. However, previous single-core cache tuning heuristics are not fully applicable to multi-core systems, which have significantly larger design spaces and additional multi-core considerations. In heterogeneous multi-core systems, the design space grows exponentially with the number of cores. Additionally, cores executing data-sharing applications cannot be tuned individually without coordinating the tuning and considering the core interactions. Cores not executing data-sharing applications could leverage single-core tuning heuristics individually, however cache tuning should not simply commence on each core simultaneously, and the number of cores being tuned should be minimized.

In a data-parallel multi-core system, applications are decomposed into equal data sets that are distributed over several cores, where each core performs the same function on that core's data set. The level of data sharing among the cores and whether or not the cores' data sets have similar cache behavior dictates the application's behavior. Two data sets have similar cache behavior if the data sets have similar miss rates when run with the same cache configuration, and thus would require the same optimal cache configuration. We note that this similarity assumption is valid because the cores are executing portions (data sets) of the same application's data.

In general, similar cache miss rates would not necessarily indicate similar cache behavior.

This application behavior can be leveraged to guide the cache tuning heuristic. For example, if an application is replicated across many cores and the cores' data sets have similar cache behavior, data sharing and core interactions do not need to be considered. In this situation, cache tuning is relatively simple since tuning could be applied to a single core's cache and the optimal configuration could be conveyed to the similarly behaving cores, thus avoiding redundant cache tuning. Alternatively, data-sharing applications where the data sets have different cache behavior may require cache tuning on several or all cores since the optimal configuration will be different across the cores. In this situation, the tuning heuristic should coordinate cache tuning among the cores to avoid simultaneously tuning all caches.

In this paper, we propose an application classification guided cache tuning heuristic for level one (L1) multi-core data caches to determine the optimal energy cache configuration. The heuristic leverages runtime profiling techniques to classify the application based on the cache behavior and data sharing. This application classification dictates the *cache tuning effort*, which includes how to explore the tunable parameters, how many cores to tune, and whether or not cache tuning should be coordinated among the cores. We quantify our heuristic's energy savings for heterogeneous 2-, 4-, 8-, and 16-core systems with highly configurable caches and evaluate energy and performance overheads incurred during cache tuning. Our heuristic searches at most 1% of the design space, yielding configurations within 2% of the optimal, and achieves an average cache subsystem energy savings of 25%.

II. Related Work

A. Multi-core Optimizations

Previous multi-core cache optimizations typically focused on improving performance. Cooperative caching [3] and proximity aware caching [1] used cache-to-cache transfers to reduce off-chip accesses and to improve cache performance, while cache partitioning [8] and scheduling heuristics [10] improved cache performance by reducing resource contention. Recently, some multi-core optimizations focused on reducing energy consumption via tuning individual cores. Merkel et al. [10] tuned individual core frequencies and co-scheduled tasks to minimize resource contention and to reduce the energy-delay product. Kumar et al. introduced a single-ISA heterogeneous multi-core architecture [7] and hill climbing tuning heuristic [6] to select cores with the best performance that minimized energy. The heterogeneous core architecture improved performance by as much as 40%, found core configurations within 5% of the optimal (best performance), searched 14% of the design space, and achieved a three fold reduction in energy consumption.

B. Runtime Single-core Cache Tuning

Configurable caches, such as the M*CORE's hardware configurable cache [9], are required for cache tuning. Zhang et al. [17] architected a highly configurable cache that used way shutdown to configure the cache size, way concatenation to configure the associativity, and fetched multiple physical cache lines to configure the logical line size. To tune this configurable cache during runtime, Zhang et al. [16] introduced a single-core L1 impact-ordered cache tuning heuristic that tuned cache parameters in order of the parameter's impact on energy (i.e., the cache size was tuned first, followed by the line size, and finally the associativity). During exploration, the cache line size and associativity were held at their smallest respective values while the heuristic increased the cache size from the smallest to the largest value in powers of two until the size increase resulted in an increase in the energy consumption. The line size and associativity were similarly tuned. This impact-ordered tuning heuristic searched 28% of the design space, achieved an average of 40% energy savings, and found cache configurations within 7% of the optimal lowest energy configuration.

TCaT [4] and ACE-AWT [5] leveraged Zhang's tuning heuristic's fundamentals for two level cache hierarchies with private and shared level two (L2) caches, respectively. TCaT searched 6.5% of the design space, found configurations within 3% of the optimal, and achieved 53% energy savings on average. ACE-AWT searched 0.2% of the design space, found configurations within 1% of the optimal, and achieved 62% energy savings on average. Since no previous tuning heuristic considered multi-core architectures and the unique multi-core tuning challenges, previous heuristics are not entirely applicable to multi-core systems, however in this paper, we leverage key fundamentals established by previous work including Zhang's impact-ordered tuning heuristic [16].

III. Runtime Multi-core Data Cache Tuning

Our runtime L1 multi-core data cache tuning heuristic leverages application classification to guide cache tuning and determines the optimal, lowest energy cache configuration. The heuristic classifies the application using cache statistics (accesses, misses, write backs, and coherence misses) gathered at runtime. These cache statistics are combined with a cache subsystem energy model (detailed in Section IV.A) to calculate the cache configuration's energy consumption and guide cache tuning. Section III.A details our target multi-core architecture and Section III.B describes our runtime application classification methodology and cache tuning heuristic.

A. Multi-core Architectural Layout

Our multi-core system consists of an arbitrary number of cores and a cache tuner, all placed on a single chip, where each core has a private, highly configurable L1 data cache [17]. We chose parameter value ranges based on our experimental results for the SPLASH-2 applications, which required optimal cache sizes ranging from 8 to 64 KB, associativities ranging from direct-mapped to 4-way, and line sizes ranging from 16 to 64 bytes. Therefore, each core's L1 data cache has a physical, size of 64 KB, which is constructed using 32 2 KB banks. The cache banks can be shutdown and/or concatenated to tune the cache size and associativity. The caches have a physical line size of 16 bytes, which can be increased by fetching multiple physical lines. The caches have

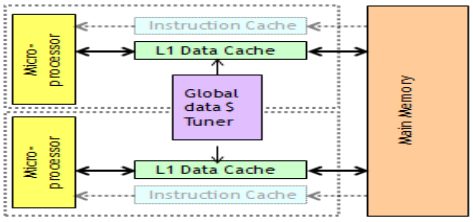

Fig. 1: Sample architectural layout for a 2-core system showing the global data cache tuner connected to each private L1 data cache.

oceannon	p0	p1	p2	p3	p4	p5	p6	p7
Miss rate (normalized to p0)	1.0	1.0	1.0	1.0	1.1	1.1	1.1	1.0

All cores have similar miss rates/cache behavior

fft	p0	p1	p2	p3	p4	p5	p6	p7
Miss rate (normalized to p0)	1.0	3.4	3.5	3.4	3.4	3.5	3.4	3.5

Cores with different miss rates/cache behavior

Fig. 2 Application classification – an example using data cache miss rates for an 8-core system where each cache is set to the base configuration

also been augmented with a small amount of custom hardware to identify coherence misses, which are misses that occur when there is a tag hit for an invalid cache block [14].

Fig. 1 depicts a sample architectural layout for a 2-core system, which contains a single, global cache tuner connected to each core's private L1 data cache (in an *n*-core system, the cache tuner connects to all *n* caches). The global tuner orchestrates the cache tuning heuristic by gathering the caches' statistics, coordinating cache tuning among the cores, and calculating the caches' energy consumption. During tuning, applications incur stall cycles while the tuner gathers cache statistics, calculates energy consumption, and changes the cache configuration. These *tuning stall cycles* introduce energy and performance overhead. Additionally, the tuning stall cycles could increase if the global tuner becomes a bottleneck while cache statistics are collected from several cores simultaneously. Our tuning heuristic considers these overheads incurred during the tuning stall cycles, and thus minimizes the number of simultaneously tuned cores and the tuning energy and performance overheads.

B. Application Classification Guided Cache Tuning Heuristic

Cache tuning is relatively simple for non-data-sharing applications where only one core's cache needs to be tuned because there are no core interactions to consider and the cores' data sets have similar cache behavior. However, tuning for data-sharing applications where the cores' data sets have different cache behavior is more complex, requiring additional tuning actions and coordinated tuning among cores. In order to determine the minimum required cache tuning effort, application classification must be done during runtime to guide cache tuning, reduce the tuning overhead, and reduce the number of simultaneously tuned cores.

Application classification determines data sharing and cache behavior at runtime using cache statistics. Coherence misses delineate data-sharing from non-data-sharing applications, where a data-sharing application's coherence misses attribute to more than 5% of the total cache misses, otherwise the application is non-data-sharing. Cache accesses

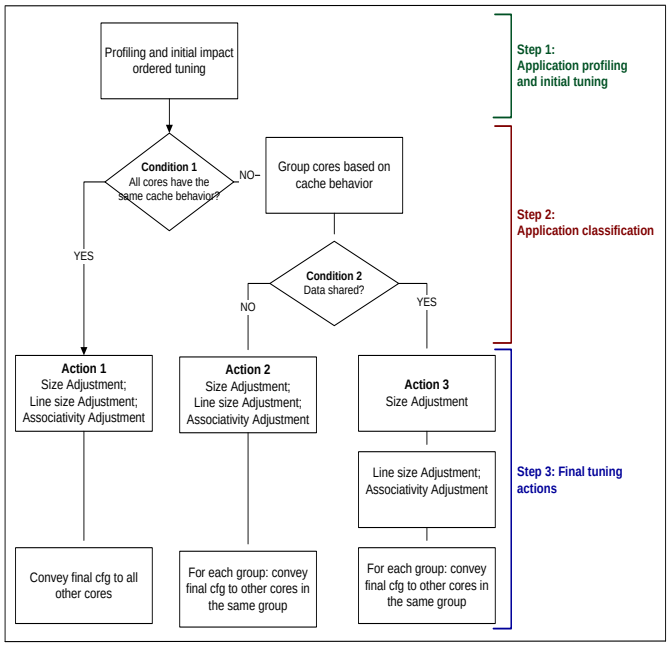

Fig. 3 Application classification guided cache tuning heuristic

and misses are used to determine if data sets have similar cache behavior. Data parallel architectures execute the same function on similar data sets. Since the cores are performing the same function on equal portions of data, data sets that have similar accesses and misses, and therefore similar miss rates, when run on caches of the same configuration are classified as having the same cache behavior. Fig. 2 illustrates these similarities using actual data cache miss rates for an 8-core system (the cores are denoted as P0 to P7) for SPLASH-2's *ocean-non* (top table) and *fft* (bottom table). We evaluate cache miss rate similarity by normalizing the caches' miss rates to the core with the lowest miss rate (P0 in this example). Since *ocean-non's* normalized miss rates are nearly 1.0 for all cores, all caches are classified as having similar behavior. *fft's* normalized miss rates show that P1 has similar cache behavior as P2 to P7 (i.e., P1 to P7's normalized miss rates are nearly 3.5), but P0 has different cache behavior than P1 to P7.

Fig. 3 depicts our application classification guided cache tuning heuristic, which consists of three main steps: 1) application profiling and initial tuning, 2) application classification, and 3) final tuning actions. Using these steps, the heuristic determines the cores' final configurations. Step 1 profiles the application to gather the caches' statistics, which are used to determine cache behavior and data sharing in step 2. Since evaluating cache behavior is most effective when the caches have the same configuration and determining data sharing is most effective when the caches are large (larger caches have more coherence misses), the heuristic initializes all of the caches to a *base configuration.* The base configuration is a 64 KB, 4-way associativity cache with a 64 byte line size. The heuristic profiles the application for one *initial* tuning interval using this base configuration.

Step 1 is critical for avoiding redundant cache tuning in situations where the data sets have similar cache behavior and similar optimal configurations. If these cache statistics indicate that all data sets have the same cache behavior, only one cache needs to be tuned, which we arbitrarily choose to

be P0. After P0's final configuration is determined, the cache tuner can immediately convey this configuration to all other caches, thus avoiding the cache tuning process on all other cores. Alternatively, if the cache statistics indicate that the cores have different cache behavior, additional cores may need to be tuned in addition to P0, however these additional cores and whether or not these cores share data cannot be determined until application classification in Step 2. Regardless of cache behavior similarities, Step 1 applies the initial impact ordered cache tuning (Section II-B) to only P0 in order to determine the initial configuration. While P0 is being tuned, the heuristic continues to identify coherence misses, which will be used to determine data sharing in Step 2.

Step 2 uses the cache behavior and coherence misses from Step 1 for application classification. Condition 1 and Condition 2 classify the applications based on whether or not the cores have similar cache behavior and/or exhibit data sharing, respectively. Evaluating these conditions determines the necessary cache tuning effort in Step 3.

Since the single-core impact ordered tuning heuristic does not consider core interactions and dependencies, P0's initial configuration determined in Step 1 is not the final configuration. Step 3 determines the final configuration using several final tuning actions that adjust the initial configuration's parameters. Step 3 leverages Step 2's application classification to determine how to perform these parameter adjustments, which cache's parameters must be adjusted, and whether or not cache tuning should be coordinated among the cores.

Cache tuning is simplified for situations with non-data-sharing applications and when all data sets have similar cache behavior, or when Condition 1 is evaluated as true. In these situations, only a single cache needs to be tuned and the heuristic performs parameter adjustments on P0 while the other cores remain fixed at the current (base) configuration. Additionally, since there is no data sharing, P0 can be tuned independently without affecting the behavior of the other cores.

The final tuning actions start with a size adjustment for P0 (Action 1). Since Step 1's initial configuration's size for P0 is typically larger than the optimal configuration's size for non-data-sharing applications, size adjustment begins with the size from Step 1 and decreases the cache size as long as decreasing the cache size decreases the energy consumption. Size adjustment is followed by similar line size and associativity adjustments, which each begin with the line size/associativity from Step 1 and increase the line size/associativity as long as increasing the line size/associativity decreases energy consumption. Finally, to complete the final tuning actions, P0's final configuration is conveyed to the other cores and the remainder of the application is executed with this final configuration.

If the data sets have different cache behavior, or Condition 1 is false, tuning is more complex and several cores must be tuned. The heuristic minimizes the number of cores that need to be tuned by grouping cores according to the data sets' cache behavior, where data sets with similar cache miss rates belong to the same group. The heuristic then tunes only one (arbitrarily chosen) cache from each group while all other cores in the group remain in the base configuration. For example, using an 8-core system and the cache miss rates in Fig. 2, *fft* has two groups: P0 belongs in one group and P1 to P7 belong in the second group. Given this grouping only P0 and P1 need to be tuned, and P1's final configuration will be conveyed to P2 to P7. Additionally, if the cores do not share data, or Condition 2 is false, the cores can be tuned independently without affecting the behavior of the other cores. The other cores chosen for tuning are set to Step 1's initial configuration and then size adjustment (decreasing the cache size) and line size/associativity adjustments (increasing line size/associativity) (Action 2) are performed on all cores identified for tuning. To complete the final tuning actions, the tuned cores convey the final configuration to the other cores in the tuned cores' respective group.

Finally, if the application shares data, or Condition 2 is true, the heuristic still only tunes one core from each group, but the tuning must be coordinated among the cores and additional configurations must be explored. Action 3 performs size adjustment on the cores identified for tuning. Since data is shared and tuning one core affects the behavior of the other cores, tuning must be coordinated. Tuning coordination requires size adjustment to complete on all cores before adjusting the remaining parameters. Additionally, instead of exploring only smaller cache sizes and larger line sizes/associativities in Step 3, the heuristic explores both smaller and larger values for each parameter since applications with shared data require additional exploration.

IV. Experimental Results

A. Experimental Setup

We quantified the energy savings and performance of our heuristic using 11 SPLASH-2 multithreaded applications (2 SPLASH-2 applications were not evaluated due to the applications' long execution times) [15] on the SESC simulator [13] for a 1-, 2-, 4-, 8-, and 16-core system. In SESC, we modeled a heterogeneous system with the L1 data cache parameters identified in Section III.A. Since the L1 data cache has 36 possible configurations, our design space is 36^n where n is the number of cores in the system. The L1 instruction cache and L2 unified cache were fixed at the base configuration and 256 KB, 4-way set associative cache with a 64 byte line size, respectively. We modified SESC to identify coherence misses.

Fig. 4 depicts the multi-core energy model used to calculate the energy consumption of each data cache configuration. Our model calculates the dynamic and static energy of each data cache, the energy needed to fill the cache on a miss, the energy consumed on a cache write back, and the energy consumed when the processor is stalled during cache fills and write backs. We gathered *dL1_misses*, *dL1_hits*, and *dL1_writebacks* cache statistics using SESC. We used CACTIv6.5 [12] to determine the dynamic cache energy dissipation for 90nm technology. We assumed the core's idle energy (*CPU_idle_energy*) to be 25% of the MIPS32 M14K processor's active energy [11] and the static energy per cycle to be 25% of the cache's dynamic energy [2].

We defined a tuning interval of 500,000 cycles, which is long enough to execute the smallest SPLASH-2 application,

fft. We looped shorter applications several times to match the execution time of the longer applications.

To simulate runtime tuning, we ran each application to completion for each configuration explored by our heuristic, calculated the total energy and performance (in cycles) using our energy model, and calculated the *configuration_energy_per_cycle*, or *total_energy/ performance*, for each configuration. We used *configuration_energy_per_cycle* to determine the energy consumed during each 500,000 cycle tuning interval and the energy consumed in the final configuration.

An application's total energy includes the energy consumed executing inferior configurations during tuning, the energy consumed executing the application in the final configuration for the remainder of the application, and the core stall energy consumed during tuning stall cycles. The tuning energy overhead is defined as the additional energy consumed when inferior configurations are executed during exploration and the core stall energy consumed during the tuning stall cycles. Energy savings were calculated by normalizing the energy to the energy consumed executing the application in the base configuration.

An application's performance includes the performance calculated by our energy model and the tuning stall cycles incurred between tuning intervals. An application's tuning performance overhead is therefore defined as *(number of configurations explored – 1) * number of tuning stall cycles*.

B. Results and Analysis

Fig. 5 (a) and (b) depict the energy savings and performance, respectively, for the optimal configuration determined via exhaustive design space exploration (optimal) for 2- and 4-core systems and for the final configuration found by our application classification cache tuning heuristic (heuristic) for 2-, 4-, 8-, and 16-core systems, for each application and averaged across all applications (Avg.). Given the exponential increase in design space size with respect to the number of cores, it was not possible to find the optimal configurations for the 8- and 16-core systems. Our heuristic achieved an average of 25% energy savings for all systems (Fig. 5 (a)) and explored at most 14 configurations—1% of the design space.

Our heuristic found the optimal configuration for 10 out of 11 applications for the 2-core system and for all 11 applications in the 4-core system. On the 2-core system, the heuristic found a final configuration within 2% of the optimal for *ocean-non*. Even though the heuristic found the optimal configuration in all but one application for the 2- and 4-core systems, executing the entire application in the optimal configuration resulted in 26% average energy savings, while the heuristic achieved 25% average energy savings. This minor energy difference is due to the tuning energy overhead. Our results showed that the largest tuning energy overhead was 4% for *radix* on the 4-core system, however, even with a 4% energy overhead, *radix* still achieved 30% energy savings.

```
total energy = Σ (energy consumed by each core)
energy consumed byeach core:
energy = dynamic_energy + static_energy + fill_energy +
         writeback_energy + CPU_stall_energy
dynamic_energy = dL1_accesses * dL1_access_energy
static_energy = ((dL1_misses * miss_latency_cycles) +
         (dL1_hits * hit_latency_cycles) +
         (dL1_writebacks * writeback_latency_cycles)) *
         dL1_static_energy
fill_energy = dL1_misses * (linesize / wordsize) *
         mem_read_energy_perword
writeback_energy = dL1_writebacks * (linesize / wordsize)
         * mem_write_energy_perword
CPU_stall_energy = ((dL1_misses * miss_latency_cycles) +
         (dL1_writebacks * writeback_latency_cycles)) *
         CPU_idle_energy
```

Fig. 4: Energy model for the multi-core system

Fig. 5 (b) shows that the average performance penalties for our heuristic for the 2- and 4-core systems were 6% and 8%, respectively, while the average performance penalties for running the application in the optimal configuration were 5% and 8%, respectively, compared to executing the application in the base cache. The tuning performance overhead due to the additional tuning stall cycles is 1% for the 2-core and less than 1% for the 4-core system. However, our results showed that, even with the tuning performance overhead, the 2- and 4-core systems achieved a 1.7x and 2.8x average speedup, respectively, compared to running the application on a single-core system.

Our heuristic achieved 26% and 25% energy savings, incurred 9% and 6% performance penalties, and achieved 4.8x and 7.9x average speedups for the 8- and 16-core systems, respectively. Although we were unable to compare these results to the optimal configuration, we estimate the tuning energy overhead as no more than 4% and the tuning performance overhead to be on average 1% for the 8- and 16-core systems based on the results for the 2- and 4-core systems.

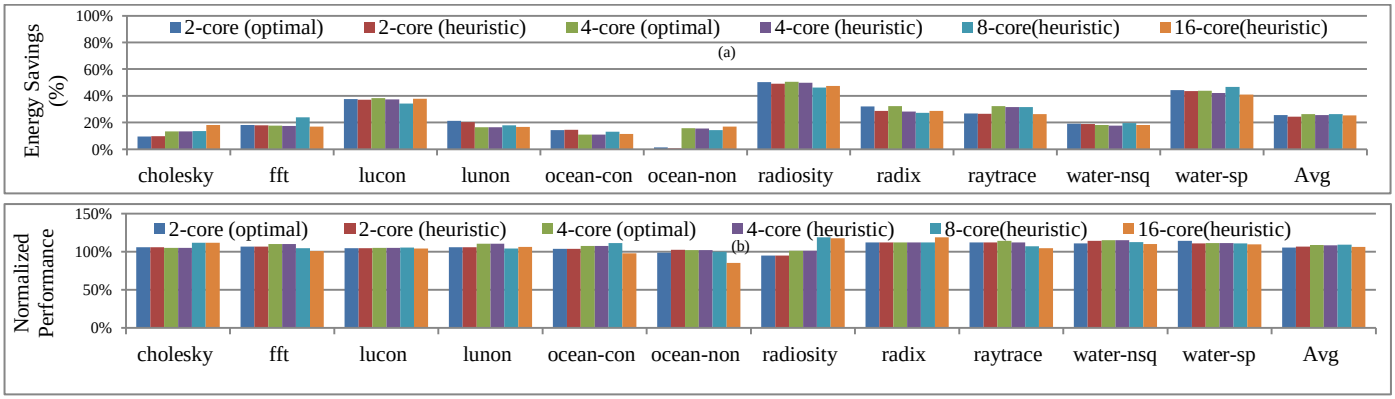

Fig. 5: (a) Energy savings and (b) normalized performance for the optimal cache (optimal) for 2- and 4-core systems and the final configuration for the application classification cache tuning heuristic (heuristic) for 2-, 4-, 8-, and 16-core systems as compared to the systems' respective base configurations.

C. Application Classification

Our heuristic classified the SPLASH-2 applications into two categories: 1) non-data-sharing applications where all cores' data sets have the same cache behavior and 2) data-sharing applications where the cores have different behaviors.

For the non-data-sharing applications, the heuristic tuned one core's cache and conveyed that cache's final configuration to the remaining caches, resulting in homogeneous final configurations across all cores. We used the optimal cache configurations, found via an exhaustive search where each core's cache could select any configuration (i.e., the cores were allowed to select heterogeneous configurations), for the 2- and 4-core systems to confirm that these applications require homogeneous final configurations. For example, all caches' miss rates normalized to nearly 1.0 for *lucon* in the 4-core system and *lucon's* optimal configurations selected via exhaustive search were homogeneous configurations where all 4 cores selected 16 KB, 4-way, 64 byte line size configurations.

Three applications *fft*, *radiosity*, and *raytrace*, were classified as data-sharing applications where the cores had different cache behavior. We observed that for these three applications one core (arbitrarily referred to as P0) typically had different behavior than the remaining cores, therefore to determine the final configuration, our heuristic tuned only P0 and one other core (arbitrarily referred to as P1), then conveyed P1's final configuration to the remaining cores, resulting in heterogeneous final configurations. For example, the miss rates for P1, P2, and P3 were nearly 3.0 times the miss-rate of P0 for *raytrace* in a 4-core system. Our heuristic selected final configurations of 16 KB, 4-way, 16 byte line size for P0 and 32 KB, 4-way, 16 byte line size for P1, P2, and P3, which is the same configuration found via an exhaustive search. Note that these three applications also share data, therefore it was necessary to coordinate data cache tuning among cores to determine the optimal configuration.

V. Conclusions and Future Work

In this paper, we presented an application classification guided cache tuning heuristic for level one data caches that found the optimal, or near optimal, lowest energy cache configuration for 2-, 4-, 8-, and 16-core systems. Our heuristic classified applications based on data sharing and cache behavior, and used this classification to identify which cores needed to be tuned and to reduce the number of cores being tuned simultaneously. Our heuristic searched at most 1% of the design space, yielded configurations within 2% of the optimal, and achieved an average of 25% energy savings. In future work we plan to investigate how our heuristic will be applicable to a larger system with hundreds of cores.

Acknowledgment

This work was supported by the National Science Foundation (CNS-0953447). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] J. A. Brown, R. Kumar, and D. Tullsen, "Proximity-aware directory-based coherence for multi-core processor architectures," *in t he Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures,* June 2007.

[2] J. Butts, and G. Sohi, "A static power model for architects," *in the Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture* (MICRO 33).

[3] J. Chang, and G. Sohi, "Cooperative caching for chip multiprocessors," in *Proceedings of the 33rd Annual international Symposium on Computer Architecture,* June 2006.

[4] A. Gordon-Ross, F. Vahid, and N. Dutt, "Automatic tuning of two-level caches to embedded applications," *in Proceedings of the Conference on Design, Automation and Test in Europe,* February 2004.

[5] A. Gordon-Ross, F. Vahid, and N. Dutt, "Fast configurable-cache tuning with a unified second-level cache," *in Proceedings of the 2005 international Symposium on Low Power Electronics and Design,* 2005.

[6] R. Kumar, D. M. Tullsen, and N. P. Jouppi, "Core architecture optimization for heterogeneous chip multiprocessors," *in Proceedings of the 15th international Conference on Parallel Architectures and Compilation Techniques,* 2006.

[7] R. Kumar, et al., "Single-ISA heterogeneous multi-core architectures for multithreaded workload performance," *in Proceedings of the International Symposium on Computer Architecture,* June 2005.

[8] C. Liu, A. Sivasubramaniam, and M. Kandemir, "Organizing the last line of defense before hitting the memory wall for CMPs," *in Proceedings of the 10th international Symposium on High Performance Computer Architecture,*February 2004.

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

[10] A. Merkel, and F. Bellosa, "Memory-aware scheduling for energy efficiency on multicore processors," *in Proceedings of the 2008 Conference on Power Aware Computing and Systems.*

[11] MIPS32 M14K http://www.mips.com/products/cores/32-64-bit-cores/mips32-m14k/

[12] N. Muralimanohar and N. P. Jouppi, "Cacti6.0 A tool to model large caches," COMPAQ Western Research Lab, 2009.

[13] P. M. Ortego and P. Sack, "SESC: SuperESCalar Simulator," http://iacoma.cs.uiuc.edu/~paulsack/sescdoc/ Dec. 2004.

[14] G. Venkataramani, et al., "Coherence miss classification for performance debugging in multi-core processors," *in the Proceedings of the 13th Workshop on Interaction between Compilers and Computer Architecture.*

[15] S. C. Woo, M. Ohara, et al., "The splash-2 programs: Characterization and methodological considerations," *in Proceedings of the International Symposium on Computer Architecture*, pp. 24–36, June 1995.

[16] C. Zhang, F. Vahid, and R. Lysecky, "A self-tuning cache architecture for embedded systems," ACM Trans. Embed. Comput. Syst. 3, 2 (May. 2004), 407-425.

[17] C. Zhang, F. Vahid, F., and W. Najjar, "A highly-configurable cache architecture for embedded systems," *30th Annual International Symposium on Computer Architecture,* June 2000.