Analysis of Cache Tuner Architectural Layouts for Multicore Embedded Systems

Tosiron Adegbija and Ann Gordon-Ross*
Department of Electrical and Computer Engineering
University of Florida, Gainesville FL, USA 32611
tosironkbd@ufl.edu and ann@ece.ufl.edu
*Also affiliated with the NSF Center for High-Performance Reconfigurable Computing (CHREC) at UF

Marisha Rawlins
Center for Information and Communication Technology
University of Trinidad and Tobago
Republic of Trinidad and Tobago
marisha.rawlins@utt.edu.tt

***Abstract*—Due to the memory hierarchy's large contribution to a microprocessor's total power, cache tuning is an ideal method for optimizing overall power consumption in embedded systems. Since most embedded systems are power and area constrained, the hardware and/or software that orchestrate cache tuning—the cache tuner—must not impose significant power and area overhead. Furthermore, as embedded systems increasingly trend towards multicore, inter-core data sharing, communication, and synchronization impose additional cache tuner design complexity, necessitating cross-core cache tuning coordination. In order to minimize cache tuner overhead, cache tuner design must consider these overheads and scalability. Whereas prior work proposes low-overhead cache tuners, scalability to multicore systems requires additional considerations. In this work, we present a low-overhead, scalable cache tuner and extensively evaluate various cache tuner design tradeoffs with respect to power and area for constrained multicore embedded systems. Based on our analysis, we formulate valuable insights and designer-assisted guidelines for modeling scalable and efficient cache tuners that best achieve optimization goals while maintaining power and area constraints.**

Keywords—cache tuning, low-power design, cache memories, multicore embedded systems, configurable hardware

I. Introduction and Motivation

Since an embedded system's memory hierarchy typically accounts for a large percentage of a microprocessor's total system power/energy [12], much emphasis has been placed on optimizing the cache subsystem's power consumption in order to achieve total system energy savings. Furthermore, due to high memory latency and memory bandwidth limitations, optimizing the cache subsystem is also critical for improving overall system performance. However, despite an embedded system's stringent design constraints (e.g., size, battery capacity, real-time deadlines, cost, etc.), there is a growing demand for performance speedups. To satisfy this growing demand, embedded system designers are increasing the number of microprocessors cores. For example, the number of cores in Samsung's Exynos microprocessor series has increased from one core in the Exynos 3 to eight cores in the Exynos 5 Octa [11]. However, increasing the number of cores significantly increases the system and optimization complexity. Thus, much research has focused on multicore embedded system cache optimizations that reduce the power consumption without significantly increasing overhead (e.g., performance, area, etc.).

Cache tuning is a common optimization method that determines the best/optimal cache configuration (specific tunable parameter values, such as cache size, associativity, and line size) that minimizes the power/energy consumption based on application requirements and design constraints. Previous work [3] showed that cache requirements vary greatly across applications and tuning the cache to a particular application can reduce average memory access energy by 62%. Cache tuning requires configurable/tunable caches [6][15], which allow parameter values to be varied and enables specialization/tuning to meet the application's requirements, and a cache tuner to orchestrate tuning (e.g., change the parameter values). Hardware and/or software cache tuners employ cache tuning heuristics/algorithms [1][3][15] to determine the best cache configuration to meet design constrains, such as reduced energy consumption. However, the cache tuner also imposes additional power, area, and/or performance overheads while exploring the configuration design space, which should be carefully considered and minimized.

Software-based cache tuners use the embedded system's processor to execute the cache tuning heuristic, which enables easy system integration, but affects the application's cache and runtime behavior due to context switching. These effects could cause the heuristic to choose non-optimal, inferior cache configurations [16].

To reduce cache tuning's impact on cache and application behavior, non-intrusive, low-overhead, hardware-based cache tuners can be used. Prior work [5] presented a hardware-based cache tuner for single-core embedded systems. In a single-core system, the cache tuner is small and lightweight, and constitutes minimal overhead, but multicore systems involve additional complexities (e.g., inter-core communication, shared resources, etc.), which could introduce significant overheads. In [10], we presented the first (to the best of our knowledge) low-overhead cache tuner for dual-core systems. While this cache tuner imposed low area, energy, and power overheads on the system, this tuner's scalability beyond two cores was not considered. As the number of cores increases, the complexity also increases, which compounds the imposed overheads. Thus, cache tuners that scale well with the number of cores without adversely impacting the embedded system's overall power consumption, area, and performance are essential to continue cache optimizations for future systems.

This paper considers three cache tuner architectural layouts with respect to scalability for multicore embedded systems: global, dedicated, and clustered cache tuners. Fig. 1 (a), (b), and (c) depict the architectural layouts for these tuners, respectively. A *global cache tuner* is a single cache tuner that tunes all of the

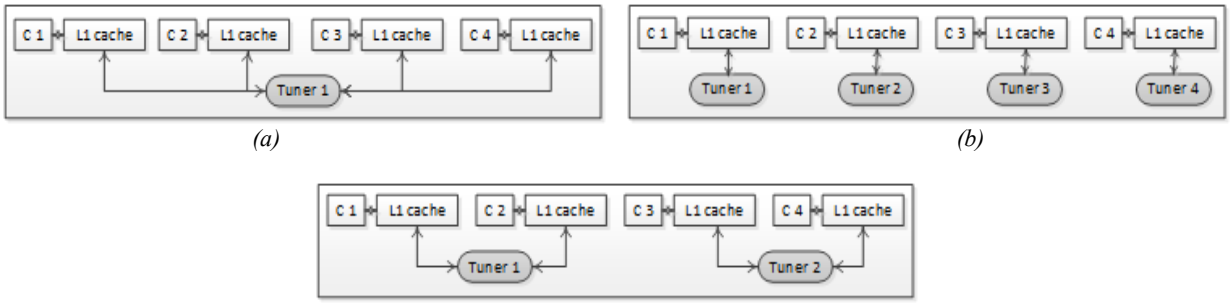


Fig. 1. Sample cache tuner architectural layout for a 4-core system with (a) global, (b) dedicated, and (c) clustered tuners

cores in the system. This cache tuner layout imposes minimal power and area overhead and is suitable for systems with a small number of cores. However, a global cache tuner may become a bottleneck in systems with a large number of cores due to this shared resource contention, resulting in significant tuning delay overhead (i.e., time waiting for the shared cache tuner). Using a *dedicated cache tuner* for each core in the system alleviates this bottleneck (e.g., an 8-core system would have eight tuners). Dedicated cache tuners reduce the shared resource contention and tuning delay, but increase the power and area overhead due to a larger number of cache tuners, and may require inter-cache tuner communication to coordinate tuning and limit avoid large power spikes if too many cores are tuning simultaneously. To trade off power, area and shared resource contention in systems with a large number of cores, a *clustered cache tuner* tunes a subset of the system's cores, and there would be several cache tuners. For example, an 8-core system could have four clustered cache tuners, each tuning two cores. However, the number of cores tuned by each clustered cache tuner could significantly impact the cache tuner's performance and overheads if the cluster size is not carefully considered.

In this work, we extend our custom cache tuner to support cache tuning in multicore embedded systems. To satisfy different design constraints, we extensively evaluate and empirically quantify the tradeoffs between global, dedicated, and clustered cache tuner architectural layouts for 2-, 4-, 8-, and 16-core systems. We evaluate the scalability of our cache tuner in these layouts as the number of cores increases with respect to the cache tuner's power consumption, area, and performance. Since the goal of using dedicated and clustered cache tuners is to reduce the shared resource contention imposed by global cache tuners, we evaluate the power consumption, area, and performance of dedicated and cluster cache tuners as compared to global cache tuners. Finally, we evaluate and quantify the power and area overheads imposed on multicore systems by the global, dedicated, and clustered cache tuners, and show that clustered cache tuners can effectively reduce shared resource contention without significant power and area overheads. We show that our cache tuner scales well with multicore systems and imposes low area and power overheads on embedded systems. Additionally, based on our design and analysis, we gain valuable insights and formulate essential design guidelines to assist designers in modeling scalable and efficient cache tuner architectural layouts in multicore embedded systems, considering tradeoffs between power, area, and performance with respect to optimization goals and design constraints.

II. Background and Related Work

Prior work has developed various configurable cache architectures and dynamic cache tuning methods to search the configuration design space. This design space contains all combinations of different tunable parameter values, and can be large for systems with many configurable parameters and parameter values (e.g., 18,000 in [4]). This section provides a brief overview of related work on configurable caches and cache tuning, which serve as background for our work.

A. Configurable Caches

Configurable caches allow for cache parameters to be tuned, enabling architectural specialization to a particular application's requirements for improved power, energy, and/or performance. Motorola's M*CORE processor [6] contained a configurable 4-way cache that provided per-way configuration using way management, which allowed the cache's four ways to be individually shutdown to reduce dynamic power during cache accesses. Modarresi et al. [8] developed a cache architecture that was partitioned and resized dynamically to improve the performance of object-oriented embedded systems.

Zhang et al. [15] developed a highly configurable cache that provided dynamic configuration of the cache's total size, associativity, and line size using small bit-width configuration registers. The proposed cache had four physical ways (i.e., the base cache was 4-way set associative) implemented as individual cache banks. The ways could be shutdown to reduce the cache size or concatenated to form a direct-mapped or 2-way set associative cache. Given a base, physical line size, the configurable cache allowed multiple physical lines to be fetched and concatenated to logically configure larger line sizes.

B. Cache Tuning

The overall power/energy savings achieved by cache tuning is strongly affected by the efficiency of the cache tuning heuristic/algorithm and how the cache tuner orchestrates the heuristic. To limit design exploration time and imposed overheads, heuristics must be efficient and effective. Zou et al. [17] proposed a configuration management heuristic to search the design space for the best cache configuration. The authors leveraged an energy-impact parameter search ordering to search the cache design space for the best cache configuration. In [10], we proposed a cache tuning heuristic that used cache statistics

combined with an energy model [15] to calculate the cache configurations' energy consumption and guide cache tuning. The heuristic determined the best cache configuration by first tuning the cache size, followed by the line size, and followed by the associativity, and stopped tuning a parameter when a parameter value change increased energy consumption. The energy model calculated the dynamic, static, fill, write-back, and processor stall energies for each cache configuration. In this work, we leverage our cache tuning heuristic and the energy model proposed in [15] for comparison purposes, however, our studies and analysis methods are independent of the specific cache tuning heuristic and/or energy model.

To facilitate cache tuning, various cache tuners have been proposed. Zhang et al. [16] designed a hardware-based cache tuner for single-core systems that dynamically tuned the cache to executing applications. In [10], we developed a low-overhead dual-core cache tuner to provide hardware support for cache tuning heuristics. Even though the cache tuners in these works were low-overhead in terms of power and area, these works did not consider scalability to larger systems with more cores, which could constitute additional cache tuning complexity and overhead. This work significantly improves on previous work by extending the dual-core cache tuner to support cache tuning in multicore systems. Additionally, we extensively study and evaluate our cache tuner's scalability in multicore systems with larger numbers of cores, where power, area, and performance tradeoffs must be carefully considered. Furthermore, unlike previous work, we carry out extensive power, area, and performance/timing analysis to quantify these tradeoffs.

III. CACHE TUNER ARCHITECTURAL LAYOUT

Typical cache tuners orchestrate the cache tuning heuristic by monitoring each explored/executed configuration's cache statistics, such as number of cache accesses, misses, etc., while the application executes for one tuning interval. To accurately evaluate each configuration, the tuning interval must be long enough to warm up the cache and stabilize the cache statistics. Using these statistics and an energy model, the cache tuner calculates each configuration's energy consumption to determine the next configuration to explore, or halts tuning if the best configuration has been determined. Thus, even though the heuristic must effectively explore the design space (i.e., limit the number of explored configurations), the cache tuner architectural layout also significantly affects the overall system power consumption, area, and tuning delay. This section details the configurable cache architecture considered in this work, the cache tuner architectural layouts, and our hardware implementations.

A. Configurable Cache Architecture

In our analysis, we consider a multicore system with an arbitrary number of cores on a single chip, where each core has a private, highly configurable level one (L1) data cache [15]. Since we only evaluate the L1 data cache and there is no shared level two cache, the L1 instruction cache's configuration is arbitrary. Each cache[1] has a physical size of 32 Kbyte, which models a typical, current embedded systems microprocessor (e.g., ARM Cortex-A7 MPCore [2]). Each cache consists of sixteen 2 Kbyte banks that can be individually shutdown and/or concatenated to tune the cache size and associativity, resulting in cache sizes range from 2 Kbyte to 32 Kbyte and associativities range from direct-mapped to 4-way set associative. Each cache has a physical line size of 16 bytes, which can be logically increased by fetching multiple lines, resulting in line sizes ranging from 16 to 64 bytes. Given these parameter values, the design space contains 36 different configurations. Even though this design space is smaller than prior work, our fundamental analysis and discussions are applicable to any larger design space or additional configurable parameters, such as L1 instruction caches, shared last level caches, etc.

B. Global, Dedicated, and Clustered Tuners

Fig. 1 (a) depicts a sample 4-core system with a single global cache tuner, which connects to each core's private L1 cache. In an *n*-core system, the global cache tuner connects to all *n* cores' caches. The global cache tuner gathers cache statistics from all of the cores, coordinates tuning between cores, and calculates each core's cache energy consumption. While the global cache tuner gathers cache statistics, calculates power/energy consumption, and changes the cache configuration, application execution is typically stalled, thus incurring tuning power and performance overhead. Since the global cache tuner is a shared resource bottleneck, the tuning delay in number of stall cycles an application experiences increases as the number of cores increases. For example, our experiments showed a 344% increase in tuning delay for a 16-core system as compared to an 8-core system (Section IV).

Dedicated cache tuners reduce the tuning delay, since each cache tuner only tunes a single core's cache. Fig. 1 (b) depicts a sample 4-core system with dedicated cache tuners, each of which connect to the associated core's private L1 cache. Dedicated caches tuners use the system's existing communication architecture for inter-cache tuner communication if tuning must be coordinated between cores (e.g., due to data sharing between the cores), depending on the applications' requirements, which may impose tuning delay. Since most of the communication occurs between the cache tuner and the associated cache, dedicated cache tuners typically do not constitute significant communication traffic on the system's communication architecture. However, since the number of cache tuners scales linearly with the number of cores, the area and power overheads are approximately *n* times greater than a global cache tuner for an *n*-core system.

Clustered cache tuners trade off area and shared resource contention in large systems by tuning only a subset of caches. Fig. 1 (c) depicts a sample 4-core system with two clustered cache tuners, where each cache tuner tunes two cores' caches. When using clustered cache tuners, an architectural layout decision is required since the cache tuners can connect to

[1] Any future reference to cache implicitly refers to the L1 data cache only unless otherwise noted.

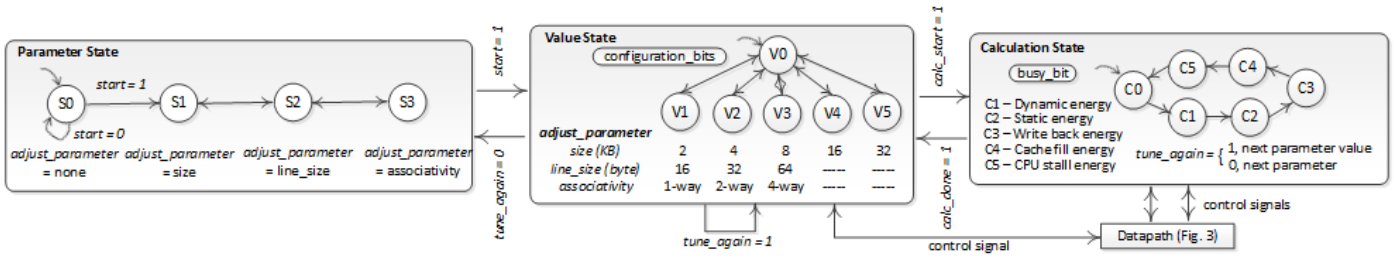


Fig. 2. Hierarchical state machine for the cache tuners

several possible cluster sizes. For example, a 16-core system could consist of cluster sizes of two, four, or eight, with eight, four, or two clustered cache tuners, respectively. Fewer clustered cache tuners impose less power and area overhead, but increase the shared resource contention and tuning delay, since each cache tuner must tune more cores. Alternatively, more clustered cache tuners impose more power and area overhead, but reduce the shared resource contention and tuning delay. Thus, the cluster size must be carefully selected to maximize optimization potential.

C. Hardware Implementation

We implemented our cache tuner in multiple architectural layouts and for multiple cores to evaluate the architectural layout options for systems with up to sixteen cores. However, the basic structure of the cache tuner is similar for all the architectural layouts. All cache tuners use a hierarchical state machine to explore the design space and control the datapath that performs the energy calculations. The global cache tuner has a single datapath that is shared by all the cores, the dedicated cache tuner's datapath is only used by cache tuner's associated core, and the clustered cache tuner's datapath is shared by only the cluster's associated cores. In this subsection, we describe our cache tuner's basic state machine and datapath structures.

1) State Machine

Fig. 2 depicts the hierarchical state machine, which contains three sub state machines: the *parameter, value,* and *calculation* states. Since the cores may choose different cache configurations and for the global and clustered cache tuners, multiple cores may be tuning simultaneously, the cache tuner contains per-core configuration bits, which control the cache configuration, and parameter and value states. Since the calculation state contains the energy calculation datapath hardware, which can impose large area overhead if replicated per core, to reduce area requirements, all cores share a single calculation state, however, this shared resource imposes tuning delay. We evaluate the tradeoff between area and tuning delay using clustered cache tuners in Section IV.

The parameter state changes the parameter being tuned. The initial state *S0* represents the cache tuner idle state, wherein the associated core is not currently tuning, *adjust_parameter = none*, and initializes variables that are used with the cache statistics to calculate the energy consumption in the calculation state. When a new application is executed, tuning begins and *start* = 1, which sets *adjust_parameter* to designate the parameter being tuned in the sub-states, *S1*, *S2*, and *S3*, as *size, line_size,* and *associativity*, respectively, and triggers a transition to the value state.

The value state changes the value of the parameter being tuned using six sub-states, and the sub-states' actions are dependent on *adjust_parameter*'s designation. State *V0* uses the *tune_again* signal from the calculation state to determine when to change the parameter's value (*tune_again* = 1), and when to change the parameter being tuned (*tune_again* = 0, i.e., all values for *adjust_parameter* have been evaluated). State *V0* also sets the configuration bits that determine the actual parameter values and changes the cache configuration based on energy calculations from the calculation state. When *adjust_parameter = size,* states *V1* through *V5* change the size to 2, 4, 8, 16, and 32 Kbyte, respectively; when *adjust_parameter = associativity*, states *V1* through *V3* change

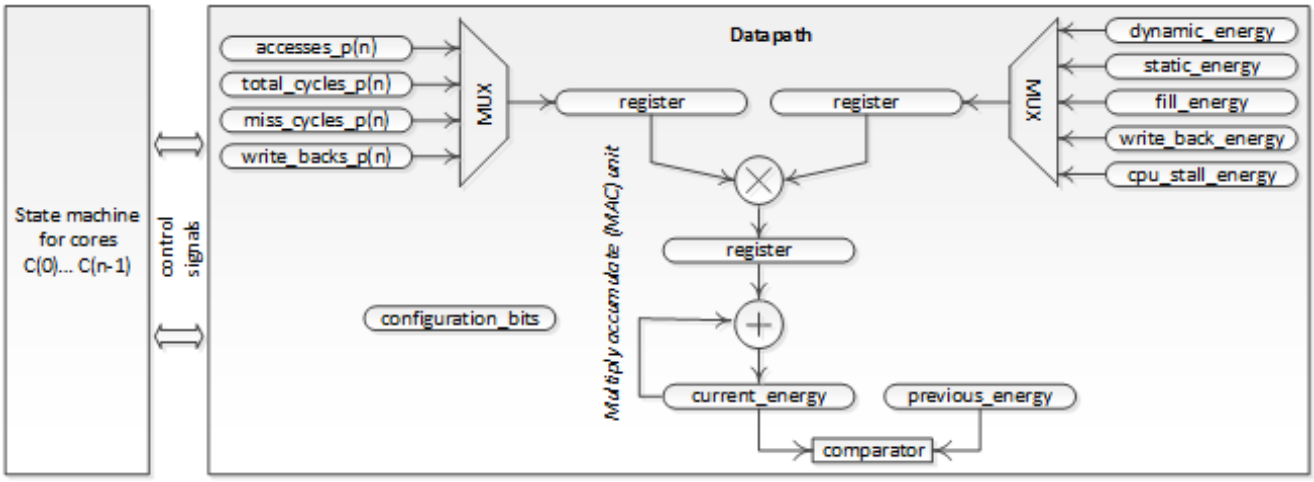


Fig. 3. Cache tuner datapath for energy calculations. The datapath is replicated for all of the tuners.

the associativity to 1-, 2-, and 4-way, respectively; when *adjust_parameter = line_size,* states *V1* through *V3* change the line size to 16, 32, and 64 bytes, respectively. After the parameter value is changed, the application executes for one tuning interval. After the tuning interval, states *V1-V5* set *calc_start = 1,* which triggers a transition to the calculation state.

The calculation state interfaces with the datapath using multiple control signals and calculates the energy consumption based on the cache statistics and the energy model [15] using six sub-states. When *calc_start = 1*, state C0 sets the *busy_bit* to ensure exclusive use of the calculation state by a single core. To ensure atomic calculations, once the *busy_bit* is set, the *busy_bit* can only be cleared after the *calc_done* signal is set to 1. States *C1* through *C5* calculate the dynamic, static, CPU stall, write back, and cache fill energies, respectively, using the energy model. After the calculations, state *C0* changes the *tune_again* signal and the *calc_done* signal, clears the *busy_bit* to 0, and transitions the state machine from the calculation state back to the value state.

2) Datapath for Energy Calculation

Fig. 3 depicts the datapath for energy calculation, which uses a multiply-accumulate unit (MAC) to calculate the total energy consumption. For brevity, we only show a global cache tuner's datapath, which is shared all cores. Dedicated and clustered cache tuners would contain one datapath for each core or cluster, respectively. Multiplexers, which are set by the current calculation state, select the specific cache statistic and energy values to multiply while calculating the dynamic, static, CPU stall, write back, and cache fill energies, depending on the current calculation state, and these intermediate values are accumulated to calculate the total energy consumption.

The datapath uses 32-bit registers to store the per-core cache statistics—total cache access, total cycles, miss cycles, and write backs—during each tuning interval's execution. The 32-bit registers are sufficient to store the cache statistics without saturation considering the tuning interval length, and the number of registers required depends on the cache configuration design space. Given a tuner shared by *n* cores, there are *n* sets of cache statistic registers. During the tuning interval, the datapath snoops the cache operations to record these statistics and store energy values. The datapath contains 36 16-bit registers to store pre-determined (Section IV.A) dynamic energies for all the 36 possible cache configurations and five registers to store the static energies for the 2 Kbyte, 4 Kbyte, 8 Kbyte, 16 Kbyte, and 32 Kbyte caches. Since there are three possible cache line sizes, and different line sizes each consume different cache fill and write back energies, three registers each are used to store the cache fill and write back energies, and one register is used to store the CPU stall energy. Even though larger design spaces would require additional registers, the tuners' basic architectural layout and functionality are independent of the cache configuration design space.

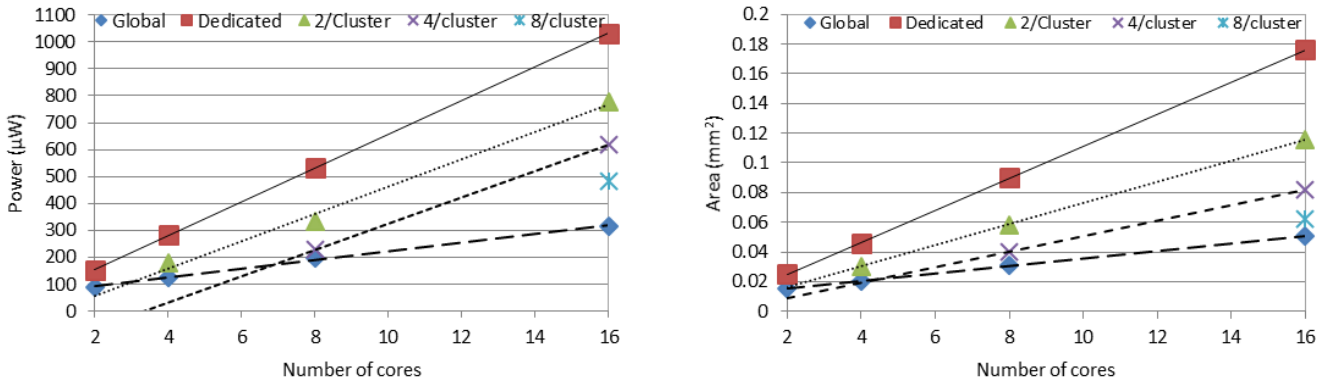
To guide the design space exploration, the datapath uses two per-core 32-bit registers to store the value of the energy consumption of the prior interval's cache configuration, *previous_energy,* and the current interval's energy consumption. The cache tuning heuristic compares *previous_energy* to *current_energy* to determine the final energy value from the calculation state. If *current_energy* is less than *previous_energy*, *tune_again* is set to 1 in state *C0*. Otherwise, *tune_again* is 0 and *previous_energy* remains unchanged, implying that the previous cache configuration consumes less energy than the current cache configuration.

Finally, a per-core 11-bit *configuration_bits* register stores the cores' cache configurations by controlling way shutdown, way concatenation, and line size adjustment (Section IIA) to change the cache's configuration during tuning. From each 11-bit register, five bits represent the cache sizes, and three bits each are used to represent the associativities and line sizes.

IV. Experimental Results

A. Experimental Setup

To quantify the power and area tradeoffs of the different cache tuner architectural layouts, we implemented and extensively evaluated global and dedicated cache tuners for 2-, 4-, 8-, and 16-core systems. For the clustered cache tuners, we implemented and evaluated 2-core clusters for the 4-, 8-, and 16-core systems (i.e., one tuner for every two cores), 4-core clusters for the 8- and 16-core systems, and 8-core clusters for



(a) *(b)*

Fig. 4. (a) Power consumption and (b) area trends for the global, dedicated, and clustered cache tuners (*n*/cluster, where *n* is the cluster size) as the number of cores increases from two to sixteen cores.

the 16-core systems. We did not evaluate a clustered 2-core system since that layout is equivalent to a global cache tuner. We assumed that the core's threads/applications were independent, and thus the cores could be tuned independently, however, our future work will consider tuning dependencies.

We modeled all the cache tuner architectural layouts in synthesizable VHDL, and synthesized the layouts using Synopsys Design Compiler [13] and the Synopsys 90nm Generic Library to quantify the cache tuners' power consumptions, areas, and tuning delays as the number of cores increases. Since cache tuner's architectural layout does not affect design exploration (i.e., each layout explores the configurations in the same order), the performance overhead incurred during design space exploration is consistent across all layouts. However, the architectural layouts could impose additional overhead on the tuning intervals by increasing the tuning stall cycles, during which energy consumption is calculated, the next configuration is chosen, the cache configuration is changed, and the cache contents are flushed (if necessary).

We quantified the overhead incurred by the total tuning stall cycles using eleven benchmarks from the SPLASH-2 benchmark suite [14]. We used the SESC simulator [9] to gather cache statistics and calculated *total_tuning_stall_cycles* as (*number of configurations explored – 1*) * *layout's tuning stall cycles*. TABLE 2 depicts all the benchmarks used and the number of configurations explored using our cache tuning heuristic (Section II.B). TABLE 1 depicts the number of tuning stall cycles for the different number of cores and cache tuner architectural layouts. We assumed a clock frequency of 2 GHz and a tuning interval of 500000 cycles.

B. Power, area, and tuning delay with respect to the number of cores

Fig. 4 (a) and (b) depict the power consumption and area trends, respectively, for the global, dedicated, and clustered cache tuners (*n*/cluster, where *n* is the cluster size) as the number of cores increased. The results depict a nearly linear increase in the power and area as the number of cores increased, revealing good scalability for all architectural layouts to future systems.

Fig. 4 (a) shows that for every power-of-two increase in the number of cores increased the global cache tuner's power consumption by 49% on average. Even though all cores shared a single global cache tuner, additional cores introduced a constant number of additional registers and logic to preserve per-core state machine information, configuration bits, and energy calculations. The dedicated and clustered cache tuners' power consumptions increased more rapidly than the global cache tuners as the number of cores increased. On average, every power-of-two increase in the number of cores increased the dedicated and clustered cache tuners' power consumptions by 89% and 111%, respectively, due to the increase in number of cache tuners in the system.

Similarly, Fig. 4 (b) shows that the cache tuners' area increased similarly to the power consumption as the number of cores increased for all architectural layouts. On average, every power-of-two increase in the number of cores increased the global, dedicated, and clustered cache tuners' areas by 51%, 93%, and 100%, respectively.

TABLE 2. BENCHMARKS AND NUMBER OF CONFIGURATIONS EXPLORED FOR 2-, 4-, 8-, AND 16-CORE SYSTEMS

Benchmark	2-core	4-core	8-core	16-core
cholesky	15	14	14	14
fft	14	14	14	14
lucon	11	14	14	14
lunon	10	15	14	14
ocean-con	10	15	12	15
ocean-non	14	14	14	15
radiosity	12	14	14	15
radix	10	10	13	13
raytrace	18	16	16	16
water-nsquare	14	15	14	17
water-spatial	14	17	16	15
AVERAGE	13	14	14	15

TABLE 1. ARCHITECTURAL LAYOUTS' TUNING STALL CYCLES

	Global	Dedicated	2/cluster	4/cluster	8/cluster
2-core	266	258			
4-core	322	258	266		
8-core	326	258	266	322	
16-core	1446	258	266	322	326

Results also showed that tuning delay scaled well for the dedicated and clustered cache tuners and increased steadily for the global cache tuner as the number of cores increased. The results showed (details omitted for brevity) that every power-of-two increase in the number of cores increased the global cache tuner's tuning delay by 121% on average. Increasing the number of cores did not affect the dedicated and clustered tuners' tuning delays, since each tuner's cores remained unchanged.

C. Tuning delay analysis

To quantify the importance of reducing the tuning bottleneck imposed by the global cache tuner, we compared the dedicated and clustered cache tuners' tuning delay to the global cache tuner. Fig. 5 depicts the cache tuners' tuning delays normalized to the global cache tuner for 2-, 4-, 8-, and 16-core systems. As compared to the global cache tuner, dedicated caches tuners reduced the tuning delay by 3%, 20%, 21%, and 82% for the 2-, 4-, 8-, and 16-core systems, respectively; the 2/cluster cache tuner by 17%, 18%, and 82% in the 4-, 8-, and 16-core systems, respectively; the 4/cluster cache tuner by 1% and 78% in the 8- and 16-core systems, respectively; and the 8/cluster cache tuner by 77% in the 16-core system. Even though dedicated tuners alleviated the global cache tuner's bottleneck, these results show that clustered tuners can also significantly reduce these bottlenecks, especially when small clusters are used in large systems (e.g., 2/cluster in a 16-core system). For example, compared to the dedicated cache tuner, the 2/cluster and 4/cluster cache tuners' average tuning delays increased by only 2% and 12%, respectively, thus making clustered cache tuners a viable tradeoff between tuning delay and area/power overheads.

Fig. 5. Tuning delay normalized to the global cache tuner for the dedicated and clustered cache tuners (*n*/cluster, where *n* is the cluster size) as the number of cores increases from two to sixteen cores.

D. Power and area analysis with respect to the global tuner

To quantify the power and area overheads imposed by dedicated and clustered cache tuners with respect to the global tuner, we compared the dedicated and clustered cache tuners' power consumption and area to the global cache tuner. Fig. 6 (a) and (b) depict the increase in power consumption and area, respectively, of the dedicated and clustered cache tuners normalized to the global cache tuner. Fig. 6 (a) shows that the dedicated, 2/cluster, 4/cluster, and 8/cluster cache tuners increased the power consumption by an average of 149%, 86%, 56%, and 53%, respectively, as compared to the global cache tuners. In summary, the power overhead with respect to the global tuner reduced as the number of cores per tuner increased, thus the global cache tuner's power scales well with increased number of cores.

Fig. 6 (b) shows that the dedicated, 2/cluster, 4/cluster, and 8/cluster cache tuners increased the area by an average of 156%, 87%, 44%, and 23%, respectively, as compared to the global cache tuners, showing similar scalability as the power consumption. Dedicated cache tuners imposed significant power consumption and area overheads, but clustered cache tuners significantly reduced the power consumption and area without significantly increasing the tuning delay. For example, compared to dedicated cache tuners in the 16-core system, clustered cache tuners reduced the power consumption and area on average by 40% and 51%, respectively, with an average tuning delay increase of only 3%. Thus, clustered tuners serve as a good tradeoff for power consumption, area, and tuning delay, especially in large system with several cores.

E. Overheads imposed by tuning stall cycles

To quantify the overheads imposed by *total_tuning_stall_cycles* on different benchmarks, we calculated *total_tuning_stall_cycles* imposed by the architectural layouts on different benchmarks using the number of configurations explored (TABLE 2) and the tuning stall cycles (TABLE 1). For brevity, we omit the graphs and only report the average results for the 16-core system. As expected, the global cache tuner in the 16-core system imposed the maximum number of additional cycles across all architectural layouts. On average over all the applications, the global cache tuner imposed a *total_tuning_stall_cycles* of 19,850 cycles. The dedicated, 2/cluster, 4/cluster, and 8/cluster tuners imposed *total_tuning_stall_cycles* of 3,542 cycles, 3,652 cycles, 4,421 cycles, and 4,476 cycles, respectively. Thus, relative to the 500,000 cycle tuning interval, the cache tuners impose a maximum overhead of 4%, on average over all the benchmarks.

F. Power and area overheads with respect to microprocessors

To quantify the power and area overheads imposed by the cache tuners on microprocessors, we evaluated the power and area overheads of the architectural layouts with respect to the MIPS32 M4K 90nm processor [7]. Since the MIPS32 M4K is a small, low power processor that consumes 12mW of power at 200MHz and has an area of 0.21mm^2, our evaluations are pessimistic. We estimated the power consumption and area for 2-, 4-, 8-, and 16-core systems based on the MIPS32 M4K, assuming a linear increase in power consumption and area as the number of cores increased. Fig. 7 (a) and (b) depict the percentage power and area overheads, respectively, for global, dedicated, and clustered cache tuners in 2-, 4-, 8-, and 16-core systems. On average over all the systems, the global cache tuners imposed power and area overheads of 0.5% and 4.73%, respectively. Dedicated cache tuners increased the average power and area overheads to 1.16% and 11.03%, respectively. For global and dedicated cache tuners, results showed that the average power and area overheads *decreased* as the number of cores increased, since the microprocessor's area increased as the number of cores increased. Thus, dedicated cache tuners scale well as the number of cores increases, and can be used as an alternative to a global cache tuner to reduce the tuning delay without significant power and area overheads. Overall, these results show that our cache tuners constitute minimal power and area overheads on a microprocessor.

We also observed that while more cores per cluster (e.g., 4/cluster, 8/cluster, etc.) reduced the power and area overheads, the 2/cluster cache tuners reduced shared resource contention in large systems' cache tuners without significant power and area overheads. However, in systems where power consumption and area must be prioritized over shared resource contention (e.g.,

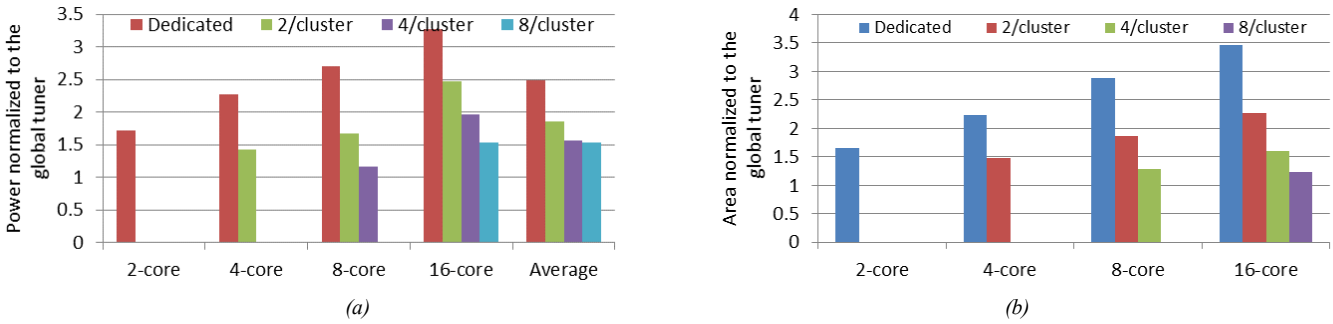


Fig. 6. (a) Power consumption and (b) area normalized to the global cache tuner for the dedicated and clustered cache tuners (*n*/cluster, where *n* is the cluster size) as the number of cores increases from two to sixteen cores.

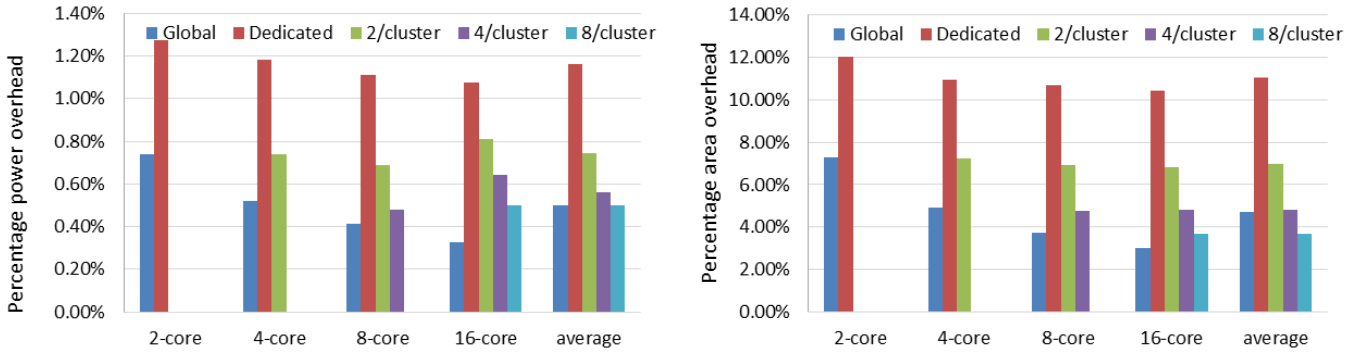


Fig. 7. Percentage (a) power and (b) area overheads with respect to the MIPS M4K processor

on a low-power chip with advanced communication networks that can easily be leveraged by the cache tuners), more cores per cluster (e.g., 4/cluster, 8/cluster) may be used to further reduce the power and area overheads.

V. Conclusions and Future Work

Cache tuning specializes a system's cache configurations to executing applications to increase optimization potential. In power and area constrained embedded systems, the cache tuner must minimize the imposed power, area, and tuning delay overheads to fully realize optimization potentials. Since multicore embedded systems introduce additional system complexity, cache tuner design in multicore systems also increases in complexity. In this paper we presented a low-overhead cache tuner that scales to multiple cores and extensively evaluated various cache tuner architectural layouts—global, dedicated, and clustered cache tuners—for multicore embedded systems. We evaluated the cache tuners' power consumptions and areas as the number of system cores increases, and quantified the overhead imposed by these tuners in a power and area constrained embedded system's microprocessor. Our results show that our cache tuner constitutes low performance, power, and area overheads. Additionally, our results showed that in large systems (e.g., 16-core systems), using clustered cache tuners with a few core per cache tuner could be used to reduce the shared resource contention as compared to a global cache tuner without significant power and area overheads, thus precluding the need for private cache tuners.

Our future work includes evaluating cache tuner designs in more complex embedded systems, such as high performance embedded systems with up to 128 cores and systems with data sharing between the cores, where the tuning dependencies must be carefully considered. We also plan to explore other options for reducing the cache tuning overhead, such as incorporating a custom lightweight communication network for cache tuners to make the tuning process independent of the on-chip communication architecture/traffic.

Acknowledgments

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 authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] T. Adegbija and A. Gordon-Ross, "Exploiting dynamic phase distance mapping for phase-based tuning of embedded systems," IEEE International Conference on Computer Design, October 2013.

[2] Cortex-A7 Processor – ARM, http://www.arm.com/products/processors/cortex-a/cortex-a7.php

[3] A. Gordon-Ross and F. Vahid, "A self-tuning configurable cache," IEEE Design Automation Conference, July 2007.

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

[5] A. Gordon-Ross, P. Viana, F. Vahid, W. Najjar, and E. Barros, "A one-shot configurable-cache tuner for improved energy and performance," Design, Automation, and Test in Europe, April 2007.

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

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

[8] Modarressi, S. Hessabi, and M. Gourdarzi, "A reconfigurable cache architecture for object-oriented embedded systems," Canadian Conference on Electrical and Computer Engineering, 2006.

[9] P. Ortega and P. Sack, "SESC: superscalar simulator," http://iacoma.cs.uiuc.edu/~paulsack/sescdoc December 2004.

[10] M. Rawlins and A. Gordon-Ross, "A cache tuning heuristic for multi-core architectures," IEEE Transactions on Computers, Vol. 62, Issue 8, August 2013.

[11] Samsung Exynos, http://www.samsung.com/exynos/

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

[13] Synopsys Design Compiler, Synopsis Inc. www.synopsys.com.

[14] S. C. Woo, et al., "The splash-2 programs: characterization and methodological considerations," International Symposium on Computer Architecture, June 1995.

[15] C. Zhang, F. Vahid, and W. Najjar, "A highly-configurable cache architecture for embedded systems," International Symposium on Computer Architecture, May 2003.

[16] C. Zhang, F. Vahid, and R. Lysecky, "A self-tuning cache architecture for embedded systems," ACM Transactions on Embedded Computing Systems, May 2004.

[17] X. Zou, J. Lei, and Z. Liu, "Dynamically reconfigurable cache for low power embedded systems," International Conference on Natural Computation, August 2007.