Application-Aware Adaptive Cache Architecture for Power-Sensitive Mobile Processors

GARO BOURNOUTIAN and ALEX ORAILOGLU, University of California, San Diego

Today, mobile smartphones are expected to be able to run the same complex, algorithm-heavy, memory-intensive applications that were originally designed and coded for general-purpose processors. All the while, it is also expected that these mobile processors be power-conscientious as well as of minimal area impact. These devices pose unique usage demands of ultra-portability but also demand an always-on, continuous data access paradigm. As a result, this dichotomy of continuous execution versus long battery life poses a difficult challenge. This article explores a novel approach to mitigating mobile processor power consumption while abating any significant degradation in execution speed. The concept relies on efficiently leveraging both compile-time and runtime application memory behavior to intelligently target adjustments in the cache to significantly reduce overall processor power, taking into account both the dynamic and leakage power footprint of the cache subsystem. The simulation results show a significant reduction in power consumption of approximately 13% to 29%, while only incurring a nominal increase in execution time and area.

Categories and Subject Descriptors: B.8.0 [**Performance and Reliability**]: General; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Cellular architecture*; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*

General Terms: Design, Performance

Additional Key Words and Phrases: mobile processors, application-aware, low-power cache design, dynamic, power-sensitive

ACM Reference Format:
Bournoutian, G. and Orailoglu, A. 2013. Application-aware adaptive cache architecture for power-sensitive mobile processors. ACM Trans. Embedd. Comput. Syst. 13, 3, Article 41 (December 2013), 26 pages.
DOI: http://dx.doi.org/10.1145/2539036.2539037

1. INTRODUCTION

The prevalence and versatility of mobile processors has grown significantly over the last few years. At the current rate, mobile processors are becoming increasingly ubiquitous throughout our society, resulting in a diverse range of applications that will be expected to run on these devices. Modern mobile processors are required to be able to run algorithmically-complex, memory-intensive applications comparable to applications originally designed and coded for general-purpose processors. Furthermore, mobile processors are becoming increasingly complex in order to respond to this more diverse application base. Many mobile processors have begun to include features such as multilevel data caches, complex branch prediction, and now even multicore architectures, such as the Qualcomm Snapdragon and ARM Cortex-A9.

Authors' addresses: G. Bournoutian (corresponding author) and A. Orailoglu, Computer Science and Engineering Department, University of California, San Diego, 9500 Gilman Dr. MC 0404, La Jolla, CA 92093-0404; email: {garo, alex}@cs.ucsd.edu.

DOI: http://dx.doi.org/10.1145/2539036.2539037

It is important to emphasize the unique usage model embodied by mobile processors. These devices are expected to be always on, with continuous data access for phone calls, texts, emails, internet browsing, news, music, video, TV, and games. In addition, remote data synchronization, secure off-device data storage, and instant application deployment all promote this cloud style of computing. Furthermore, these devices need to be ultra-portable, able to be carried unobtrusively on a person, and require infrequent power access to recharge.

With the constraints embodied by mobile processors, high performance, power efficiency, better execution determinism, and minimized area are typically the most pressing concerns. Unfortunately, these characteristics are often adversarial, and focusing on improving one often results in worsening the others. For example, to increase performance, a more complex cache hierarchy is added to exploit data locality, but introduces larger power consumption, more data access time indeterminism, and increased area. However, if an application is highly regular and contains an abundance of both spatial and temporal data locality, then the advantages in performance greatly outweigh the drawbacks. On the other hand, as these applications become more complex and irregular, they are increasingly prone to thrashing. For example, video codecs, which are increasingly being included in wireless devices like mobile phones, utilize large data buffers and significantly suffer from cache thrashing [Lee et al. 2004].

In particular, in mobile phone systems, where power and area efficiency are paramount, smaller, less-associative caches are typically chosen. Earlier researchers realized that these caches are more predisposed to thrashing and proposed solutions such as the victim cache [Jouppi 1990] or dynamically-associative caches [Bournoutian and Orailoglu 2008] to improve cache hit rates. While these approaches can mitigate cache thrashing and result in improved execution speed and some dynamic power reduction, the aggressive, all-day usage model of mobile processors demands much larger reductions in overall processor power to help improve the battery life of these devices. Furthermore, static leakage power is becoming increasingly more predominant as feature size continues to be reduced to 45 nm and lower and can be as much as 30–40% of the total power consumed by the processor [Rodriguez and Jacob 2006; ITRS 2009]. Since caches typically account for 30–60% of the total processor area and 20–50% of the processor's power consumption, they are ideal candidates for improvement to help reduce overall processor power [Malik et al. 2000].

In this article, we propose a novel approach to deal with the unique constraints of mobile processors. While normally the processor operates in a baseline configuration wherein the battery life may be muted in order to deliver full execution speed and responsiveness, a user may, on the other hand, desire to sacrifice a negligible amount of that execution speed to help prolong the overall life of the mobile device. The fundamental goal of our approach is that this sacrifice of speed be disproportionately related to the amount of power saved, allowing large gains in cache power savings without significantly disrupting execution performance. In particular, we found the L2 (and optional L3) cache exhibited unbalanced access patterns, wherein large portions of the cache may be unused at a given time, while other portions may be more heavily used and subject to thrashing.[1] By adapting dynamically to the application's runtime behavior, we can intelligently make modifications to the cache to appreciably reduce static power while minimally affecting miss rates and associated dynamic power consumption. We leverage application-specific information to help tailor the configuration of the hardware to best serve the given application's memory needs. We show the

[1]While this proposal could also be applied to the L1 cache, the relatively small size and high access rates would greatly diminish possible power improvements. Thus, for the purposes of this proposal, we focus on the larger L2 cache, which is present in most modern smartphones.

implementation of this architecture and provide experimental data taken over a general sample of complex, real-world applications to show the benefits of such an approach. The simulation results show improvement in overall processor power consumption of approximately 13% to 29%, while incurring a minimal increase in execution time and area.

2. RELATED WORK

In the last five years, the industrial mobile smartphone processor space has seen enormous expansion. Processors provided by companies such as ARM, Samsung, and Qualcomm have become increasingly more powerful and complex and are used in a wide variety of industrial applications. For example, current smartphone technology often incorporates a mixture of ARM9, ARM11, and ARM Cortex embedded processors, along with a number of sophisticated specialized DSP processors, such as Qualcomm's QDSP6. These mobile phones are expected to handle a wide variety of purposes, from remote data communication to high-definition audio/video processing, and even live multiplayer gaming. These target applications are becoming increasingly more complex and memory intensive, and numerous techniques have been proposed to address the memory access challenges involved.

Common structural techniques rely on segmenting the word- or bit-lines to reduce latency and dynamic power. Subbanking [Ghose and Kamble 1999] divides the data arrays into smaller subgroups, and only the "bank" that contains the desired data is accessed, avoiding wasted bit-line precharging dissipation. The Multiple-Divided Module (MDM) Cache [Ko et al. 1995] consists of small, stand-alone cache modules. Only the required cache module is accessed, reducing latency and dynamic power. Unfortunately, these techniques do not address the significantly increasing static leakage power.

Phased caches [Hasegawa et al. 1995] first access the tag and then the data arrays. Only on a hit is the dataway accessed, resulting in less dataway access energy at the expense of longer access time. Similar to the aforementioned structural techniques, leakage power is not addressed, and in fact often becomes worse due to the increase in access time.

Filter caches [Kin et al. 1997] are able to reduce both dynamic and static power consumption by effectively shrinking the sizes of the L1 and L2 caches, but suffer from a significant decrease in performance. This large amount of performance degradation is typically unacceptable in modern mobile processors.

Similarly, on-demand selective cache ways [Albonesi 1999], which dynamically shut down parts of the cache according to application demand, suffer from sharp performance degradation when aggressively applied. Similarly, Speculative Way Activation [Calder et al. 1996; Inoue et al. 1999] attempts to make a prediction of the way where the required data may be located. If the prediction is correct, the cache-access latency and dynamic power consumption become similar to that of a direct-mapped cache equivalent. If the prediction is incorrect, the cache is accessed a second time to retrieve the desired data. This results in some additional latency, as the predicted way must be generated before data address generation can occur, and there is still the significant amount of static leakage power that is not addressed.

Configurable caches [Gordon-Ross et al. 2009] also focus on reducing both dynamic and static power consumption by employing way concatenation and way shutdown to tune the entire cache's size and associativity. The proposal relies on a multiphase online heuristic for exploring a search space of approximately 18,000 possible cache configurations, taking into account the separate L1 instruction and data caches, as well as a unified L2 cache with way management. While the results appear encouraging, the overhead in terms of power and area of this on-chip dynamic tuning heuristic are

not included and would likely countervail much of the cache power improvements. Furthermore, the results utilized a small 64KB L2 cache and assumed only 10% of the total cache drain stemmed from static leakage. Most mobile smartphones utilize larger L2 caches, and the static leakage typically dominates the total cache power due to nanometer technology scaling [Rodriguez and Jacob 2006]. Given that the proposed way shutdown mechanism only allows reducing the number of ways down to one (not completely disabling the cache index), there can still be substantial static leakage when cache indices are underutilized.

Smart caches [Sundararajan et al. 2011] also employ adaptive techniques to reduce both dynamic and static power consumption. While the results are positive, the approach targets high-end general-purpose processors with out-of-order execution. Thus, the results are based on an L2 cache size of 2MB, which is much larger than those commonly found in smartphones. Furthermore, the assumption of an out-of-order processor would hide the impact of any increases in cache miss rates (due to disabling sections of the cache), which would otherwise have led to a more significant performance degradation. On the other hand, many mobile smartphones currently opt for in-order processors due to their smaller area and power footprint.

There are also several techniques that specifically attempt to target leakage power within caches. The Gated-V_{dd} technique [Powell et al. 2000] allows SRAM cells to be turned off by gating the supply voltage away from the cell, effectively removing leakage but also losing all the state within that cell. The Data Retention Gated-Ground (DRG) Cache [Agarwal et al. 2002] reduces the leakage power significantly, while still retaining the data within the memory cells while in standby mode. On the other hand, there is a significant increase to the word-line delay of 60% or higher depending on the feature size. Similarly, the Drowsy Cache [Flautner et al. 2002] provides a low-power "drowsy" mode, where data is retained but not readable, and a normal mode. Leakage power is significantly reduced when in drowsy mode, but there is a cost and delay to switching between drowsy and normal mode. Since the cache is primarily present to mitigate the performance implications of long memory latencies, it is important to avoid causing significant degradation in performance just to recuperate leakage power. There is an important balance that must occur, where the combined dynamic and leakage power are reduced while not significantly degrading performance (since having a longer runtime will ultimately lead to still more dynamic and leakage power).

3. MOTIVATION

A typical data-processing algorithm consists of data elements (usually part of an array or matrix) being manipulated within some looping construct. These data elements each effectively map to a predetermined row in the data cache. Unfortunately, different data elements may map to the same row due to the inherent design of caches. In this case, the data elements are said to be in "conflict." This is typically not a large concern if the conflicting data elements are accessed in disjoint algorithmic hotspots, but if they happen to exist within the same hotspot, each time one is brought into the cache, the other will be evicted, and this *thrashing* will continue for the entire hotspot.

Given complex and data-intensive applications, the probability of multiple cache lines being active within a hotspot, as well as the probability of those cache lines mapping to the same cache set, increases dramatically. As mentioned, much prior work has already gone into minimizing and avoiding cache conflicts and thrashing in order to improve overall performance and reduce dynamic power usage within the cache subsystem.

While we may encounter cache conflicts and thrashing in certain areas within the cache, the rest of the cache may have ample capacity or remain idle for large periods of time. As mentioned, static leakage power within these caches is also becoming increasingly prohibitive. In particular, the larger secondary (L2) cache contributes the

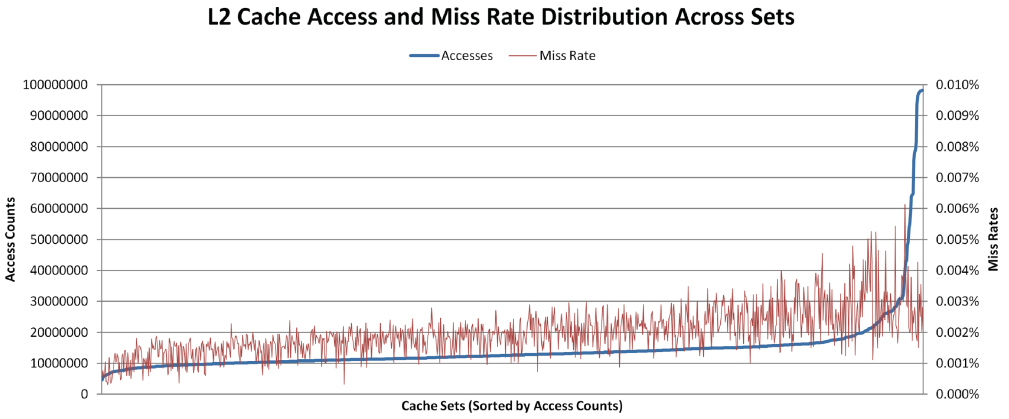

Fig. 1. L2 cache access and miss-rate distribution for *h264dec*.

majority of the leakage power compared to the primary (L1) cache. One would like to eliminate as much leakage power as possible, without significantly degrading the normal performance of the memory subsystem. Fortunately, the L2 cache exhibits unbalanced access patterns, since most memory accesses are serviced by the L1 cache. Thus, typically only a sporadic number of memory accesses cascade into the L2 cache. Because of this observation, much of the L2 cache is idle for extended periods of time. We can take advantage of this inactivity and temporarily shut down those idle cache sets to eliminate the associated leakage power. As the application progresses, the locations within the cache that are active or idle may change, and thus the architecture must also take this into account.

To illustrate this observation, Figure 1 provides a sorted distribution of L2 cache accesses per cache set for the *h264dec* video decoder benchmark. As one can see, the number of times a given cache set is accessed varies across a large range. Some cache sets are heavily accessed, while some are very rarely accessed. Furthermore, the figure also provides the miss-rate contribution for each cache set. One important correlation is that those cache sets that are rarely accessed also contribute the least to the aggregate miss rate (since they are accessed less frequently).

The goal of this article is the dynamic identification of those cache sets that are highly utilized and prone to thrashing, as well as those cache sets that are rarely utilized and are contributing needlessly to leakage power, and to adjust the cache accordingly to help conserve power while also preserving, or even improving, performance. In order to achieve this goal, an efficient hybridization of information from both compile-time and runtime analysis is essential. The compiler can help glean crucial generalized application characteristics that would otherwise be too costly to obtain at runtime. The hardware can then utilize this application-specific information and react more intelligently.

Using the aforementioned observation regarding unequal access patterns within the power-hungry L2 cache, it becomes evident that certain portions of the cache are overutilized, while other portions are underutilized. This imbalance can degrade both performance and power and should be rectified. Ideally, one would like to automatically increase the storage available for crowded locations of the cache, as these locations are more prone to thrashing during heavy utilization within a temporally short period of time. Similarly, when locations are idle over a period of time, they should automatically be disabled to reduce leakage power. One challenge is at what resolution to track and respond to these access patterns. For simplicity, large regions of the cache can be

grouped together and treated en masse. This reduces the complexity necessary for the hardware detection and correction logic but imposes a coarse-grained reaction to cache behavior. If a situation arose where one portion of the cache group was heavily utilized while another portion of the same logical group was highly underutilized, the system would not be able to satisfy both constraints. On the other hand, using a fine-grained hardware approach would allow greater flexibility, even if neighboring locations within the cache had widely divergent access patterns.

Another challenge is that the activity within the various locations of the cache changes throughout the program's lifetime. At one point in time, a given location may be highly utilized, but then at some later interval, it could be idle. Obviously, the hardware can dynamically detect these changes in activity level, but current levels may not necessarily foretell future patterns. All that is needed is some way to prime the hardware about such future patterns.

Indeed, application-specific information can be leveraged to inform the hardware about probable memory access patterns. In this manner, the hardware can intelligently make decisions based on general application characteristics, in addition to real-time access patterns. To be frugal, coarse-grained application-specific information can be used to tailor the hardware's behavior. Instead of leveraging detailed instruction or basic-block characteristics, a more global application signature is proposed. Conceptually, this will greatly simplify the hardware storage and complexity; there is no need to query the program counter to determine which instruction or basic block the processor is currently executing. Instead, generalized application attributes are mined that reflect the memory access patterns and frequencies for the overall program. This more coarse-grained resolution may not be able to react as precisely as the fine-grained approach. On the other hand, a drastically smaller data footprint is required when compared to a per-instruction or basic-block approach.

In order to extract this application-specific information, profiling is leveraged. The target application is executed and profiled using one or more representative program inputs. The real-time access patterns are observed and extracted, generating a signature reflective of the application that was run. An alternative approach to profiling would be to use pure static analysis to garner application-specific information. While static analysis can formally generate a similar signature that is more immune to individual program inputs, it suffers when pointers and dynamic memory are used within the application. Such memory accesses are commonly indeterminate statically and can only be resolved at runtime. Thus, profiling would be more adept at extracting accurate application characteristics in these cases.

Using these concepts, an architecture is proposed that will leverage application-specific information to help guide a dynamically reconfigurable cache that is capable of detecting and responding to imbalances in access frequencies that would degrade either performance or power.

4. PROPOSAL

The proposed solution consists of three elements: a compiler-driven application memory analysis, a runtime cache access behavior monitoring mechanism, and a dynamic cache expansion and contraction mechanism. A high-level architectural overview is shown in Figure 2. Initially, an application is profiled offline to garner memory characteristics and create a small application signature that will be embedded within the application binary. This data will convey coarse-grained information about the memory access patterns seen during profiling. Next, while the application is running on the processor, the cache will maintain runtime statistics, keeping track of access frequency rates to each of the cache sets. Lastly, using the application-specific information in conjunction with the real-time access statistics, the cache will detect whether a modification is

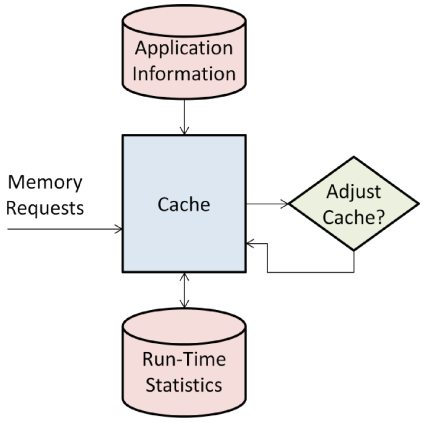

Fig. 2. High-level architectural overview of proposed design.

merited for any given cache sets. If a cache set would benefit from having increased capacity, the cache will adapt by providing more storage locations. On the other hand, if the cache set could effectively be shrunk and merged into another set, the cache will adapt accordingly and disable the unneeded portions to reduce leakage power.

As will be shown in detail in the following section, a pseudo-associative cache design is employed wherein cache access to a given cache set can be redirected into a secondary location within the same cache [Agarwal and Pudar 1993]. When additional capacity is required, this secondary set can also be accessed in order to expand the available storage locations for a particular address index. In contrast, if the current capacity is more than sufficient and it is determined that we can maintain the same performance without as much storage for the given location, it is folded into its secondary location. This action effectively halves the amount of storage available, since it now shares the same storage as the secondary location. The proposed architecture will dynamically detect these situations during runtime and make the appropriate adjustments to the cache to effectively use the available storage and also avoid wasteful leakage power for cells that remain idle. The next section will describe in detail how this high-level architecture is implemented.

5. IMPLEMENTATION

The proposed solution enables two complementary types of behavior to occur per cache set: *expansion* and *contraction*. A set-associative cache architecture is assumed throughout this article, wherein each cache set (as opposed to each individual line) will be annotated with a small number of additional bits to keep track of runtime behavior. In this manner, the storage and logic overhead is independent of the level of associativity, and indeed this same architecture would also work for a direct-mapped cache. Figure 3 provides a high-level view of the architectural additions. In particular, a small shift register is added to each cache set and will be used to measure the access frequency of that set. Upon accessing the cache set, the *frequency shift register* (FSR) is left shifted and fed a least-significant-bit (LSB) value of 1. Upon the decay signal, determined by a *global countdown register* (DECR), the FSR is right shifted and fed a most-significant-bit (MSB) value of 0. In this manner, the FSR will saturate with all 1's if highly accessed, saturate with all 0's if rarely accessed, or otherwise possess the property of having a continuous run of 1's of some length L starting from the LSB. This structuring of the FSR will minimize bit-flipping transitions (avoiding needless dynamic power consumption) and will greatly reduce the complexity of comparing

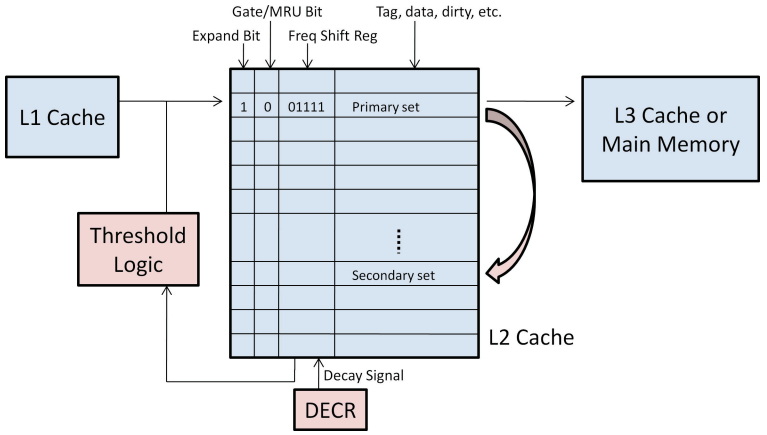

Fig. 3. High-level cache implementation.

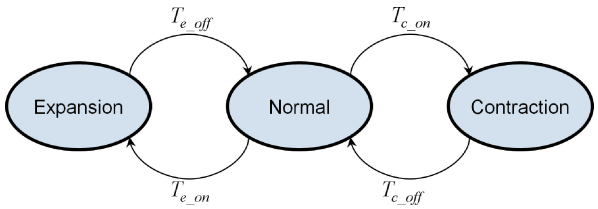

Fig. 4. Cache behavior state diagram.

the value in the FSR with a given threshold value. Additionally, the FSR values are initialized to all 0's upon reset or flush.

The architecture also has four global threshold registers: T_{e_on} (*Expansion On*), T_{e_off} (*Expansion Off*), T_{c_off} (*Contraction Off*), and T_{c_on} (*Contraction On*). These threshold registers are the same size as the FSR's and will contain a single 1 in a specific bit position to indicate its threshold. Thus, the comparison of a threshold with the FSR is simply a combinational *AND* fed into an *OR*-reduction (i.e., if any bit is a 1, then the result is 1, else 0). If the FSR has met or exceeded a given threshold, it can quickly and efficiently be detected, and the cache can then make the appropriate changes to either enable or disable expansion or contraction, based on the particular threshold value(s) met.

Figure 4 provides the basic state diagram of this cache behavior. The threshold registers are constrained in the following manner in order to avoid ambiguity and deadlock.

$$T_{e\_on} > T_{e\_off} \geq T_{c\_off} > T_{c\_on}. \tag{1}$$

Thus, the minimum size of the threshold registers (and also the FSR) is three bits, and we will denote this size with N.

To implement this adaptive cache architecture, one will need three extra bits for every cache set (one for the *expand bit*, one for the *MRU bit*, and one additional tag bit to differentiate values from primary versus secondary locations). The decay countdown register (DECR) is 64 bits, plus a 64-bit reset value register that is loaded into the DECR upon reaching 0. Furthermore, there is the addition of the N bits used for the FSR on each cache set and the N bits for the four threshold registers. Since we use an eight-bit FSR ($N = 8$) and have 1,024 sets in our implementation, this results in

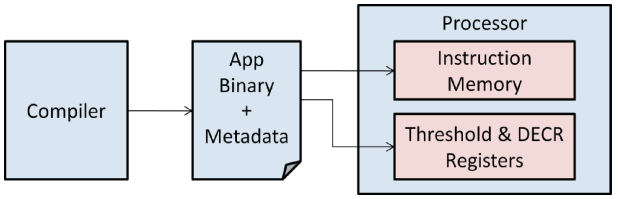

Fig. 5. Overview of compiler and hardware interaction.

an additional 1,428 bytes (≈1.39 KB) added to our L2 cache. This additional storage overhead is negligible compared to the actual cache size of 256 KB.

The details of the compiler-driven determination of threshold values, as well as the expansion and contraction mechanisms, are described in detail in the following three sections.

5.1. Compiler-Driven Threshold Selection

Our initial work [Bournoutian and Orailoglu 2010] assumed fixed, predefined threshold values, yet not all applications exhibit the same memory access behaviors. By using application-specific information, we enable a more fine-grained capability for matching the cache behavior with a given application, yielding improved power and performance benefits compared to a system-wide fixed-value approach. To accomplish this, we need to strengthen the interaction between the compiler and the underlying hardware microarchitecture. Inherently, there are certain global application characteristics that can more easily be ascertained during compile-time analysis and profiling, such as average cache miss rates and time between cache accesses. This information would be prohibitively costly and complex to generate on the fly during runtime. On the other hand, there are certain behavioral aspects of the application that can only be ascertained during runtime, such as actual cache hit/miss behavior based on user input or real-time application data streams. These events cannot statically be known a priori or guaranteed with profiling alone. Given this trade-off between information available at compile-time versus runtime analysis, it becomes evident that some mixture of the two will be essential. The compiler will help glean crucial generalized application characteristics that would otherwise be too costly to obtain at runtime, embedding this information in the application binary. Then, upon the operating system loading the application into the processor, the *loader* can convey these values to the underlying hardware microarchitecture by initializing the four threshold registers and the decay counter reset register to appropriate values for the given application. An overview of this architecture is shown in Figure 5.

To accomplish this analysis, we will leverage offline application profiling using a customized implementation of Valgrind [Nethercote and Seward 2007] in order to extract global information regarding cache access behavior for the given application. We are interested in seeing how the memory for the application behaves: how frequently we access cache sets, how often we encounter a cache miss, and how often we access the same cache set temporally. Using such information, we can create a signature for the program that represents the generalized memory behavior for the application. In particular, we will capture the following pieces of global cache information.

—Miss rate (MR).
—Access rate[2] (AR).

[2]Access rate refers to the percentage of total instructions executed that accessed the cache.

Table I. T_{e_on} Mapping Conditions

MR Range	T_{e_on} Value
$0.400 \leq MR$	00000100
$0.200 \leq MR < 0.400$	00001000
$0.100 \leq MR < 0.200$	00010000
$0.050 \leq MR < 0.100$	00100000
$0.025 \leq MR < 0.050$	01000000
$MR < 0.025$	10000000

—Temporal delay[3] mean (TD_{μ}).
—Temporal delay standard deviation (TD_{σ}).

Using the aforementioned profiling information, we can calculate values for the threshold registers. Given that we have N bits for the threshold registers and the constraint embodied by Equation (1), T_{e_on} can contain a 1 only in the leftmost $N-2$ positions. Thus, we have $N-2$ possible values for T_{e_on}. Our goal is to correlate T_{e_on} with the miss rate (MR), where larger miss rates warrant a lower (right-shifted) threshold. Thus, we will be more predisposed to enable expansion in this situation, even if access frequencies are not very high. To determine an appropriate T_{e_on} value, we will map the $N-2$ possible values into exponentially decreasing ranges of miss rates. As most applications typically have lower L2 cache miss rates, this exponential arrangement allows for more fine-grained resolution at the lower miss rates while still handling the occurrence of a higher miss rate. Since N will typically be small to conserve area and power, there is only a modest number of possible threshold representations. To provide a continuum on these threshold values that handle a diverse range of miss rates and correspond to given access frequencies and decay periods, an exponential distribution is needed.

We will need to specify a maximum miss rate (MR_{max}) to begin this mapping. Based on MR_{max}, we will determine the logarithmic normalization coefficient:

$$MR_{\lambda} = \frac{2^{N-2}}{MR_{max}}. \tag{2}$$

Then, using MR_{λ}, we can map the profiled MR into a corresponding T_{e_on}:

$$T_{e\_on} = \begin{cases} 1 \ll (N - \lfloor \log_2[(MR)(MR_{\lambda})] \rfloor), & \text{if } MR \geq (2/MR_{\lambda}), \\ 1 \gg (N-1), & \text{if } MR < (2/MR_{\lambda}). \end{cases} \tag{3}$$

Based on profiling the memory patterns of the various application benchmarks, we found that a setting of $N = 8$ provided sufficient resolution within our FSR to capture runtime cache access behavior and allow for the differentiation of eight unique levels of utilization. Additionally, we define $MR_{max} = 0.40$, as a 40% miss rate is a reasonable upper bound; were an L2 cache to encounter more than 40% miss rates, it would be indicative of a more systemic problem necessitating the cache size to be increased or greater associativity be used. Given this, Table I demonstrates the mapping logic for translating the profiled MR into a T_{e_on} value.

While the previous calculations effectively incorporate observed miss-rate behavior, the actual impact of cache accesses relative to the application as a whole is not reflected. To embody the performance impact of cache misses, we need to also factor in the cache access rate (AR). Even with a large miss rate, if the access rate is quite low, then the impact of those cache misses across the entire application is muted. Thus, we propose

[3]Temporal delay refers to the number of cycles between accesses to a given cache set.

Table II. T_{c_on} Mapping Conditions

TD_μ Range	T_{c_on} Value
$TD_\mu < 6.25e2$	00000001
$6.25e2 \leq TD_\mu < 1.25e3$	00000010
$1.25e3 \leq TD_\mu < 2.50e3$	00000100
$2.50e3 \leq TD_\mu < 5.00e3$	00001000
$5.00e3 \leq TD_\mu < 1.00e4$	00010000
$1.00e4 \leq TD_\mu$	00100000

a further modification of T_{e_on} in the following manner.

$$T_{e\_on} = \begin{cases} T_{e\_on} \gg 2, & \text{if } 10\% \leq AR, \\ T_{e\_on} \gg 1, & \text{if } 5\% \leq AR < 10\%, \\ T_{e\_on}, & \text{if } AR < 5\%. \end{cases}$$

To ensure the satisfaction of Equation (1), if T_{e_on} gets shifted too far right (e.g., if the 1 is located in one of the two rightmost bit positions), it will be saturated into its lowest allowable value (0x4).

$$T_{e\_on} = 0\text{x}4, \text{ if } T_{e\_on} < 0\text{x}4.$$

Calculation of the contraction enable threshold (T_{c_on}) is done in a similar fashion. Our goal in this case is to correlate T_{c_on} with the observed average temporal access delay (TD_μ). If the delay between cache accesses is large, we want to set T_{c_on} to a higher (left-shifted) threshold. Intuitively, if the application as a whole exhibits very long periods of delay between cache accesses, we want to contract as much as possible to exploit the benefits in leakage power reduction. If a small group of cache sets are being frequently accessed (but not enough to skew the overall TD_μ), their individual FSR will allow the expansion to occur if necessary. To determine an appropriate T_{c_on} value, we will map the $N-2$ possible values into exponentially increasing ranges of temporal delays. We will need to specify a maximum temporal delay (TD_{max}) to begin this mapping. Based on TD_{max}, we will determine the logarithmic normalization coefficient.

$$TD_\lambda = \frac{2^{N-2}}{TD_{max}}. \tag{4}$$

Then, using TD_λ, we can map the profiled TD_μ into a corresponding T_{c_on}.

$$T_{c\_on} = \begin{cases} 1 \ll (\lfloor \log_2[(TD_\mu)(TD_\lambda)] \rfloor - 1), & \text{if } TD_\mu \geq (2/TD_\lambda), \\ 1, & \text{if } TD_\mu < (2/TD_\lambda). \end{cases} \tag{5}$$

For the purposes of this article, we specify $TD_{max} = 1.00e4$ based on profiling the average temporal accesses (TD_μ) across all the benchmarks. Given this, Table II demonstrates the mapping logic for translating the profiled TD_μ into a T_{c_on} value.

Finally, we need to ensure that resulting values for T_{e_on} and T_{c_on} still satisfy Equation (1). It is possible that the threshold values may violate Equation (1) by T_{e_on} not being two or more positions to the left of T_{c_on}, or vice versa. If that is the case, one of these two thresholds must be adjusted accordingly. We chose to modify T_{e_on}, to bias toward power reduction versus execution speed. This is embodied in Equation (6):

$$T_{e\_on} = T_{c\_on} \ll 2. \tag{6}$$

With regard to the disabling thresholds, T_{e_off} and T_{c_off}, they are calculated as follows.

$$T_{e\_off} = 1 \ll \log_2[T_{e\_on}] - \left\lceil \frac{\log_2[T_{e\_on}] - \log_2[T_{c\_on}] - 1}{3} \right\rceil, \tag{7}$$

$$T_{c\_off} = 1 \ll \log_2[T_{c\_on}] + \left\lceil \frac{\log_2[T_{e\_on}] - \log_2[T_{c\_on}] - 1}{3} \right\rceil. \tag{8}$$

Finally, to compute the desired interval for the decay countdown register (DECR), we leverage the information regarding the mean and standard deviation of the temporal delay. Intuitively, one would like the decay signal to occur often enough to help lower the FSR and allow idle cache sets to be contracted to reduce leakage power. On the other hand, the decay signal should not occur too fast and cause unnecessary transitions between contracted and non-contracted state. By utilizing the standard deviation of the temporal access delay, we can derive a decay interval large enough to allow average length accesses sufficient time to reaccess a cache set and counteract the decay signal if they are still in use. Assuming a Gaussian distribution, it is anticipated that the vast majority of cache sets will be accessed within the average temporal delay plus the standard deviation. Thus, if we configure the decay to occur at this interval rate, we should be able to cause cache sets being used less than this rate to begin to decay, while the other sets will not. This concept is represented in Equation (9), which defines the value for the $DECR_{reset}$ register used to reset the DECR countdown register upon reaching zero.

$$DECR_{reset} = TD_{\mu} + TD_{\sigma}. \tag{9}$$

While the emphasis of this article focuses on profiling a single application in order to generate corresponding threshold and countdown register values, it is important to briefly mention that the next generation of mobile processors are beginning to employ multicore architectures with shared L2 caches. Given this, there may now be more than one application running and concurrently accessing the L2 cache. The threshold and countdown values should take into account the class of applications that are expected to run concurrently on the system. The same profiling technique previously described can be performed on each application. Assuming all applications are prioritized equally, a simple average of the individual threshold and countdown registers can be done to determine a common set of values for all applications. On the other hand, if certain applications are more critical, the average can be weighted as needed. Yet, while having accurate threshold and countdown values helps optimize the performance and power savings, it is important to note that the system will still dynamically adjust to the actual runtime access patterns and adapt the cache configuration to best reflect the actual needs of applications currently executing on the processor.

5.2. Expansion Mechanism

This mechanism allows for the expansion of a particular cache set into a second cache set, similar to the architecture described in Bournoutian and Orailoglu [2008]. The rationale for this expansion is that the cache set in question is being heavily utilized in a temporally short period of time and thus may be prone to thrashing. This is an important difference from [Bournoutian and Orailoglu 2008], where this proposal uses access frequencies over time instead of evicted cache lines to determine whether to expand. This is detected by the current set's FSR meeting the T_{e_on} threshold, which causes the *expand bit* to be turned on. On a cache miss, if this bit is enabled, a secondary set within the same cache is accessed on the next cycle, similar to a pseudo-associative cache, as shown in Figure 3. This secondary set is determined by a fixed mapping

function based on the cache size. We investigated using an LFSR (linear feedback shift register) mapping function, as well as a basic MSB toggling function. For the LFSR approach, we chose a ten-bit LFSR with the maximal-length polynomial $x^{10} + x^7 + 1$. To determine our complement set, the current cache index is loaded into the LFSR, and the next value computed represents the complement index. The only limitation with using an LFSR is that the all-zero state will map back onto itself, essentially excluding that one particular cache set from being expanded or contracted.[4] For the MSB approach, we simply *XOR* the MSB of the index into the cache. The benefit of the LFSR approach is that the cycle length can be quite large, avoiding the situation that occurs in the basic MSB approach where the primary set maps to the secondary set and vice versa (i.e., a cycle length of 2). On the other hand, the MSB approach is economical in terms of power and area, as only a single logic gate is required, and it is a purely combinational circuit.

If the data is found within the secondary set, one effectively has a cache hit but with a one cycle penalty. If after looking into the secondary set, the data is still not located, a full cache miss occurs, and the next memory device in the hierarchy (L3 or main memory) is accessed with the associated cycle penalties. The key principle is that we only enable the secondary lookup on a per-set basis, as always enabling this secondary lookup is wasteful and can lead to worse performance when cache sets are lightly used.

Thus, we effectively can double the size of the given set without needing to double the hardware. This works in principle due to the typical locality of caches. Since our complement set is chosen to be bidirectionally distant from the current set, the probability of both the primary and complement set being active at the same temporal moment in an application is quite small. Even if this unlikely event does occur and we pollute the currently active cache set that happens to be our complement, the impact is negligible. Later in the program's progression, if the complement set is then used, it will function normally and evict the older data from the prior expansion out of its space.

Additionally, there is an *MRU bit* on each of the sets within the cache. This bit is enabled whenever a cache hit occurs on the secondary cache set; it is disabled when a cache hit occurs on the primary set. The MRU bit is also used on subsequent cache lookups to determine whether to initially do a lookup on the primary cache set or the secondary cache set, effectively encoding a most-recently-used paradigm. If the last hit occurred in the secondary set, the next access has a higher probability to also occur in the secondary set, and vice versa.

In addition, we implemented our caches to use an LRU replacement policy. When a set has its expand bit on, the LRU set (primary or secondary) is used in the event a replacement is necessary. Furthermore, for associative caches, after determining which set to use, the normal LRU rules for which way to use within that set apply. This set-wise LRU is inherently supported by the MRU bit, since there are only two possible locations.

It is important to note that once a set has its expand bit enabled, it remains enabled until the set's FSR falls below the T_{e_off} threshold. Furthermore, this expansion need not be symmetrical in the MSB approach. Even though *set-A* may have its expand bit turned on and be utilizing *set-A* and its complement *set-B*, *set-B* can still be acting as a normal non-expanded set. Only if primary accesses to *set-B* also trigger the expansion threshold will *set-B*'s expand bit also be enabled to allow expansion into *set-A*.

5.3. Contraction Mechanism

The contraction mechanism allows for the gating of idle cache sets, helping to eliminate any associated leakage power. The rationale for this contraction is that the cache set in

[4]The impact of this limitation is minimal, as it only affects one cache set out of 1,024.

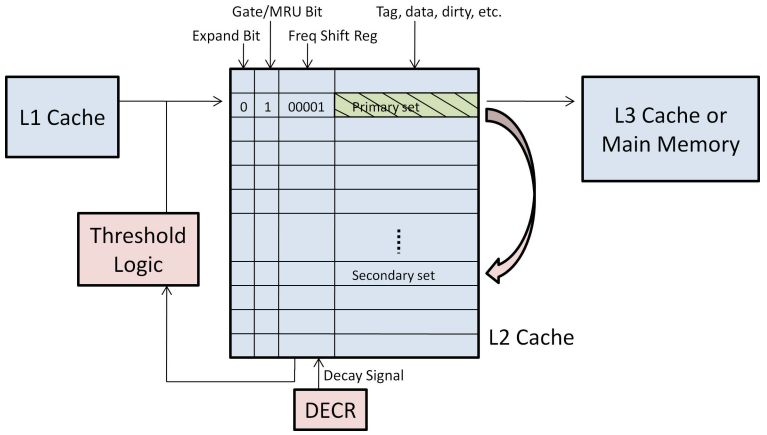

Fig. 6. Example gated primary set redirecting queries to secondary location.

question is being rarely utilized over a temporal period and thus can safely be turned off. This is detected by the current set's FSR falling below the T_{c_on} threshold, which causes the *gate bit* to be turned on. As mentioned earlier, a decay signal, determined by a clock divider, is used to periodically lower the FSR. If no accesses occur on that set, over time the decay signal will continue to lower the FSR value. On the other hand, if accesses begin to occur, they will help raise the FSR value to counteract the decay. The gate bit is the same bit as the MRU bit, but when the cache set is not "expanded", it is instead interpreted as gating the cache set and forcing all references to be redirected to the secondary set. This can safely be done, since *expansion* and *contraction* are mutually exclusive states. The secondary set is calculated in the same fashion as in the expansion state, using either the LFSR or MSB approach described earlier.

Thus, we effectively shrink the size of the given set during times of infrequent use, helping eliminate leakage power. When a cache set has the gate bit enabled, the supply voltage is cut off from the memory cells in a similar fashion to Powell et al. [2000]. This results in a reduction of 97% of the memory cell's leakage power, with only an 8% read performance overhead. Essentially, all states for the cache set other than the FSR, expand bit, and gate bit are powered off. Furthermore, if the cache set contained any dirty lines, these would be flushed serially in the background prior to the supply voltage being gated. As shown in Figure 6, any references to the gated primary set will be redirected to the secondary set. Since these locations are rarely used, it is statistically acceptable to not retain data that may currently be live within the given cache set. If the data is indeed needed at a future time, the typical cache miss penalty will be incurred. Given this, there is effectively a trade-off between aggressively gating sets and the associated negative impact with regard to performance and power resulting from having to query the next-level structure in the memory hierarchy. Appropriate values for the T_{c_on} and T_{c_off} thresholds will help balance this trade-off.

If a gated cache set begins to be accessed more frequently and its FSR meets the T_{c_off} threshold, then the gate bit will be disabled and the set will no longer be gated. Additionally, if a set enters into the expansion mode, it will force the complement set to exit contraction mode. Furthermore, if a set is in contraction mode, the complement set will not be allowed to enter contraction mode as well, to avoid deadlock. In other words, when a cache set enters contraction mode, the complement set is forced to be in *normal* mode to ensure that a memory location will exist to handle that cache address space. A state transition diagram for a complement pair of cache sets is provided in

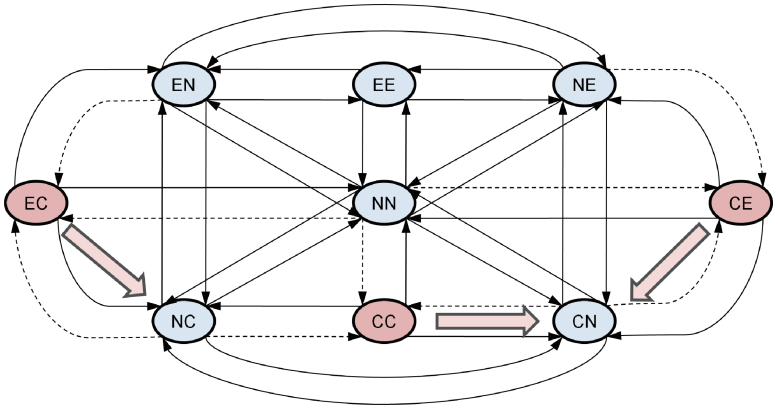

Fig. 7. State transition diagram for a pair of cache sets.

Figure 7. The letters E, N, and C indicate whether a cache set is in expansion, normal, or contraction mode, respectively. There are three disallowed states (CC, EC, and CE). They are shown in red with dashed transition lines in the figure. When one of these transitions is encountered, the state transition is corrected, as shown by the large arrows.

While gating a single cache set does help reduce some leakage power, the accumulation of a large number of sets being gated is where we begin to see tangible results. As described earlier, cache accesses within the L2 cache are quite sporadic and unevenly distributed. As will be shown in the experimental results section, there are quite a number of times during execution where many of the cache sets can safely be gated off without significantly impacting performance.

6. EXPERIMENTAL RESULTS

In order to assess the benefit from this proposed architectural design, we utilized the SimpleScalar toolset [Austin et al. 2002] using the default in-order PISA simulation engine and an eight-bit Frequency Shift Register (FSR). We chose a representative mobile smartphone system configuration, having separate 32KB L1 data and instruction caches (direct-mapped, 32-byte line size) and a unified L2 cache (four-way set-associative, 64-byte line size). We chose to use a set-associative L2 cache similar to current mobile processors; a direct-mapped L2 cache could also be used with our proposal. We chose three representative L2 cache sizes (128KB, 256KB, and 512KB), which are similar to what is currently found in smartphone processors such as the Qualcomm Halcyon MSM7230 (256KB L2), Qualcomm Blackbird MSM8660 (512KB L2), or Apple A4 (512KB L2).

The complete SPEC CINT2000 benchmark suite [SPEC 2000] is used, as well as five representative mobile smartphone applications from the MediaBench video suite [Lee et al. 1997] and the MiBench telecomm suite [Guthaus et al. 2001]. A listing of these benchmarks and their respective descriptions are provided in Table III. The benchmark's training inputs are used during profiling, while reference inputs are utilized during simulation, and all benchmarks are run to completion. The threshold and decay register values for each benchmark as determined by the compile-time application-specific profiling are shown in Tables IV, V, and VI (for 128KB, 256KB, and 512KB L2 cache sizes, respectively). For comparison, a set of hand-selected threshold and decay register values (based on previous work) for a global, pure-hardware approach is shown in Table VII. In the global approach, the same values are shared across all benchmarks

Table III. Description of Benchmarks

Benchmark	Description
bzip2	Memory-based compression algorithm
crafty	High-performance chess playing game
crc32	32-bit CRC framing hash checksum
eon	Probabilistic ray tracer visualization
fft	Discrete fast Fourier transform algorithm
gap	Group theory interpreter
gcc	GNU C compiler
gsm	Telecomm speech transcoder compression algorithm
gzip	LZ77 compression algorithm
h264dec	H.264 block-oriented video decoder
mcf	Vehicle scheduling combinatorial optimization
mpeg4enc	MPEG-4 discrete cosine transform video encoder
parser	Dictionary-based word processing
perlbmk	Perl programming language interpreter
twolf	CAD place-and-route simulation
vortex	Object-oriented database transactions
vpr	FPGA circuit placement and routing

Table IV. 128KB L2 Compiler-Generated Application-Specific Threshold Values

Benchmark	T_{e_on}	T_{e_off}	T_{c_off}	T_{c_on}	*DECR*
bzip2	00100000	00010000	00000100	00000010	2.01e3
crafty	00010000	00001000	00000010	00000001	1.33e2
crc32	00001000	00000100	00000010	00000001	4.21e2
eon	01000000	00100000	00100000	00010000	6.14e3
fft	00010000	00001000	00000100	00000010	9.04e2
gap	00001000	00000100	00000010	00000001	6.24e2
gcc	00010000	00001000	00000010	00000001	2.49e2
gsm	00001000	00000100	00000010	00000001	2.73e2
gzip	00100000	00010000	00000100	00000010	9.47e2
h264dec	00001000	00000100	00000100	00000010	8.23e2
mcf	00010000	00001000	00001000	00000100	2.81e3
mpeg4enc	00010000	00001000	00001000	00000100	1.22e3
parser	00001000	00000100	00000100	00000010	7.96e2
perlbmk	00010000	00001000	00001000	00000100	1.59e3
twolf	00001000	00000100	00000010	00000001	2.34e2
vortex	00100000	00010000	00010000	00001000	4.09e3
vpr	00001000	00000100	00000010	00000001	4.78e2

without regard to individual application characteristics. As will be shown shortly, using application-specific information to tailor the cache behavior will yield better results than an across-the-board value for the entire system.

Table VIII provides the miss-rate comparison between the baseline system and the proposed adaptive cache architecture, showing both the global threshold and application-specific threshold results. The MSB mapping function is utilized for the expansion and contraction mechanisms. As one can see, certain benchmarks result in slightly higher miss rates due to the more aggressive use of the contraction mechanism. For example, the 256KB *gzip* benchmark incurs a 5.44% degradation in L2 cache performance. But, while that may seem like a large amount, the actual miss rate is quite low (1.55%) and thus more sensitive to any deviations. Figure 8 graphically

Table V. 256KB L2 Compiler-Generated Application-Specific Threshold Values

Benchmark	T_{e_on}	T_{e_off}	T_{c_off}	T_{c_on}	*DECR*
bzip2	00100000	00010000	00000100	00000010	2.37e3
crafty	00100000	00001000	00000100	00000001	1.91e2
crc32	00001000	00000100	00000010	00000001	5.55e2
eon	00100000	00010000	00010000	00001000	6.50e3
fft	00010000	00001000	00001000	00000100	1.62e3
gap	00010000	00001000	00000100	00000010	7.38e2
gcc	01000000	00010000	00000100	00000001	3.05e2
gsm	00001000	00000100	00000010	00000001	2.88e2
gzip	10000000	00100000	00010000	00000100	2.42e3
h264dec	00010000	00001000	00000100	00000010	1.32e3
mcf	00010000	00001000	00001000	00000100	4.19e3
mpeg4enc	00100000	00010000	00001000	00000100	2.83e3
parser	00010000	00001000	00000100	00000010	1.02e3
perlbmk	00010000	00001000	00001000	00000100	2.34e3
twolf	00001000	00000100	00000010	00000001	3.66e2
vortex	00100000	00010000	00010000	00001000	5.11e3
vpr	00001000	00000100	00000010	00000001	6.02e2

Table VI. 512KB L2 Compiler-Generated Application-Specific Threshold Values

Benchmark	T_{e_on}	T_{e_off}	T_{c_off}	T_{c_on}	*DECR*
bzip2	00100000	00010000	00001000	00000100	2.83e3
crafty	00100000	00001000	00000100	00000001	1.45e2
crc32	00010000	00001000	00000010	00000001	5.99e2
eon	01000000	00100000	00100000	00010000	6.37e3
fft	00100000	00010000	00010000	00001000	4.60e3
gap	00100000	00010000	00001000	00000100	9.65e2
gcc	01000000	00010000	00000100	00000001	3.30e2
gsm	00010000	00001000	00000100	00000010	7.15e2
gzip	10000000	01000000	00100000	00010000	7.62e3
h264dec	00010000	00001000	00001000	00000100	2.43e3
mcf	01000000	00100000	00100000	00010000	7.83e3
mpeg4enc	01000000	00100000	00100000	00010000	5.37e3
parser	00100000	00010000	00010000	00001000	3.12e3
perlbmk	01000000	00100000	00100000	00010000	8.53e3
twolf	00010000	00001000	00000010	00000001	5.61e2
vortex	01000000	00100000	00100000	00010000	7.44e3
vpr	00001000	00000100	00000100	00000010	6.34e2

Table VII. Hand-Picked Global Threshold Values

Benchmark	T_{e_on}	T_{e_off}	T_{c_off}	T_{c_on}	*DECR*
ALL	10000000	00100000	00010000	00000100	2.00e3

shows the miss-rate impact across the benchmarks for the three cache configurations, compared against the baseline miss rate (100%). As expected, the application-specific approach is able to maintain or improve the miss rates, since the expansion mechanism will help alleviate cache conflicts for saturated sets, while only shrinking sets that are infrequently used. Furthermore, one can see that, on average, the *application-specific approach* (App) outperforms the *global threshold* (Glb) in terms of reducing miss rates

Table VIII. L2 Cache Miss Rates

Benchmark	Accesses	Cache Size	Baseline Miss Rate	*Global Threshold* Miss Rate	*Global Threshold* Change	*Application-Specific* Miss Rate	*Application-Specific* Change
bzip2	4.29e9	128 KB	0.2035	0.2015	0.98%	0.2015	0.98%
		256 KB	0.1699	0.1710	−0.65%	0.1710	−0.65%
		512 KB	0.1094	0.1101	−0.64%	0.1101	−0.64%
crafty	3.42e9	128 KB	0.0413	0.0420	−1.69%	0.0332	19.61%
		256 KB	0.0038	0.0040	−5.26%	0.0034	10.53%
		512 KB	0.0037	0.0038	−2.70%	0.0034	8.11%
crc32	4.26e7	128 KB	0.1919	0.1922	−0.16%	0.1785	6.98%
		256 KB	0.1750	0.1797	−2.69%	0.1419	18.91%
		512 KB	0.0944	0.0945	−0.11%	0.0899	4.78%
eon	7.45e9	128 KB	0.1166	0.1109	4.89%	0.1035	11.23%
		256 KB	0.0973	0.0991	−1.85%	0.0959	1.44%
		512 KB	0.0344	0.0351	−2.03%	0.0349	−1.45%
fft	2.48e7	128 KB	0.1591	0.1567	1.51%	0.1527	4.02%
		256 KB	0.1007	0.1049	−4.17%	0.0988	1.89%
		512 KB	0.0619	0.0620	−0.16%	0.0611	1.29%
gap	4.94e10	128 KB	0.1415	0.1423	−0.57%	0.1371	3.11%
		256 KB	0.0814	0.0818	−0.49%	0.0788	3.19%
		512 KB	0.0438	0.0437	0.23%	0.0419	4.34%
gcc	1.72e8	128 KB	0.0665	0.0679	−2.11%	0.0552	16.99%
		256 KB	0.0244	0.0252	−3.28%	0.0228	6.56%
		512 KB	0.0167	0.0169	−1.38%	0.0163	2.22%
gsm	6.00e7	128 KB	0.1673	0.1711	−2.27%	0.1629	2.63%
		256 KB	0.1222	0.1231	−0.74%	0.1227	−0.41%
		512 KB	0.0959	0.0971	−1.25%	0.0958	0.10%
gzip	3.65e9	128 KB	0.0844	0.0859	−1.78%	0.0855	−1.30%
		256 KB	0.0147	0.0159	−8.16%	0.0155	−5.44%
		512 KB	0.0113	0.0119	−5.31%	0.0118	−4.42%
h264dec	6.69e8	128 KB	0.1099	0.1121	−2.00%	0.0902	17.93%
		256 KB	0.0973	0.0991	−1.85%	0.0972	0.09%
		512 KB	0.0792	0.0795	−0.40%	0.0793	−0.15%
mcf	1.42e9	128 KB	0.1593	0.1566	1.69%	0.1389	12.81%
		256 KB	0.0992	0.1005	−1.31%	0.0997	−0.50%
		512 KB	0.0616	0.0622	−0.97%	0.0620	−0.65%
mpeg4enc	4.24e8	128 KB	0.1385	0.1369	1.16%	0.1354	2.24%
		256 KB	0.0917	0.0946	−3.16%	0.0933	−1.74%
		512 KB	0.0483	0.0492	−1.80%	0.0497	−2.83%
parser	7.65e8	128 KB	0.1104	0.1112	−0.72%	0.0883	20.03%
		256 KB	0.0814	0.0822	−0.98%	0.0820	−0.74%
		512 KB	0.0688	0.0695	−1.02%	0.0691	−0.44%
perlbmk	7.74e8	128 KB	0.1453	0.1422	2.13%	0.1374	5.44%
		256 KB	0.1067	0.1071	−0.37%	0.1065	0.19%
		512 KB	0.0601	0.0622	−3.49%	0.0599	0.32%
twolf	1.10e9	128 KB	0.1211	0.1294	−6.85%	0.1131	6.61%
		256 KB	0.1148	0.1148	−0.01%	0.1125	2.00%
		512 KB	0.0748	0.0749	−0.13%	0.0746	0.27%

Continued

Table VIII. Continued

Benchmark	Accesses	Cache Size	Baseline Miss Rate	*Global Threshold* Miss Rate	*Global Threshold* Change	*Application-Specific* Miss Rate	*Application-Specific* Change
vortex	3.34e10	128 KB	0.1554	0.1503	3.28%	0.1434	7.72%
		256 KB	0.1349	0.1351	−0.15%	0.1289	4.45%
		512 KB	0.0866	0.0870	−0.46%	0.0865	0.08%
vpr	6.92e8	128 KB	0.1606	0.1675	−4.30%	0.1615	−0.56%
		256 KB	0.1425	0.1439	−0.98%	0.1436	−0.77%
		512 KB	0.1091	0.1101	−0.92%	0.1102	−1.01%
Average	6.34e9	128 KB	0.1337	0.1339	−0.40%	0.1246	8.03%
		256 KB	0.0975	0.0989	−2.12%	0.0950	2.29%
		512 KB	0.0624	0.0629	−1.33%	0.0621	0.58%

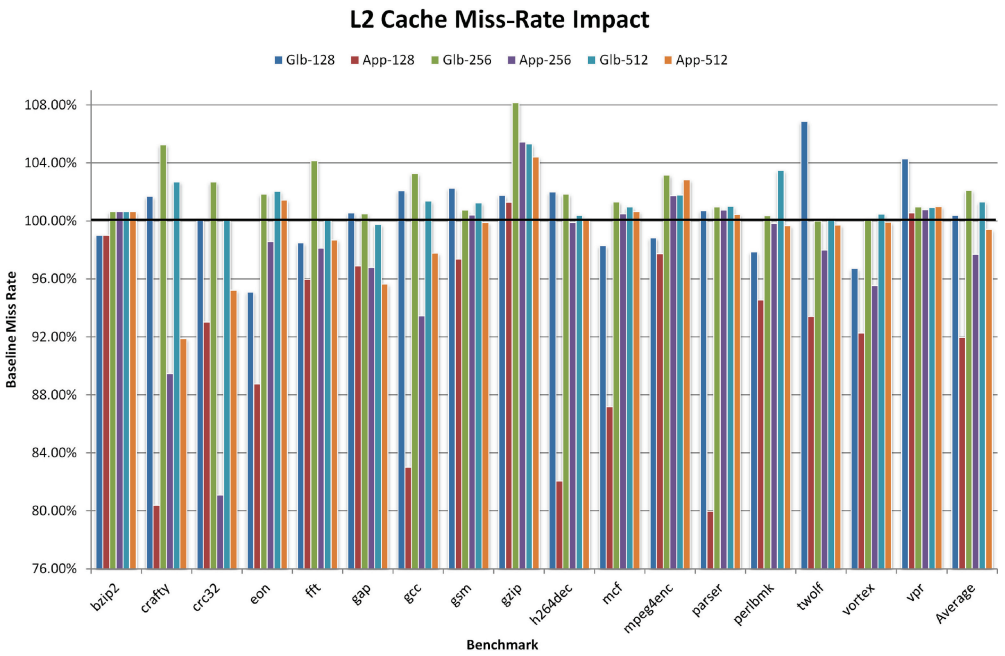

Fig. 8. L2 cache miss-rate impact.

from the baseline. Intuitively, this is also expected, since the cache behavior is tailored to more accurately respond to a particular application's memory characteristics.

As a point of comparison, Figure 9 provides the sorted distribution of L2 accesses per cache set for the *h264dec* video decoder benchmark after implementing the application-specific approach. This can be compared directly to Figure 1, which showed data before applying the proposed cache architecture. As one can see, the distribution of cache set access rates in Figure 9 has become relatively more balanced compared to the original distribution, especially for the cache sets that were originally very heavily accessed (shown on the right-hand side of Figure 1). Additionally, the expansion mechanism was able to help attenuate the misses that occurred due to overutilized cache sets encountering conflicts; the miss rate per cache set with the proposed architecture essentially has a flat trend and is centered just below 0.002%, whereas the original trend increased with the access frequency and got closer to 0.003% for the more highly-accessed sets.

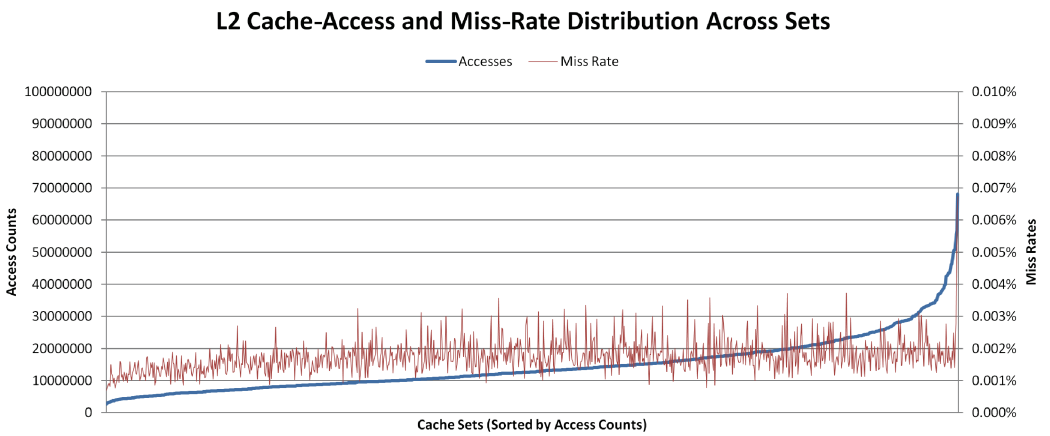

Fig. 9. L2 cache-access and miss-rate distribution for *h264dec* after using application-specific approach.

As mentioned earlier, in addition to the straightforward MSB-toggling mapping function, we also investigated using an LFSR (linear feedback shift register) approach. Whereas the MSB approach is constrained to a cycle length of 2 (i.e., primary set maps to the secondary set, and vice versa), the benefit of an LFSR mapping function is that the cycle length can grow to be quite large. We were interested to see if increasing the cycle length using an LFSR would result in a noticeable improvement to the resulting cache miss rates compared to the simple MSB approach. Table IX provides 256KB cache miss-rate improvements over MSB when using an LFSR with a cycle period of 1,023 (i.e., ten-bit maximal length sequence). As one can see, the improvements are extremely minuscule, being less than a thousandth of a single percent, and in some instances, even worse than the MSB approach. Given this and the associated timing and logic overhead to implement the LFSR mapping function, the MSB implementation appears to be more efficient, as it only requires a single-bit *XOR* gate and is purely combinational. Thus, we use the MSB mapping function for the remainder of our calculations.

Luckily, the slight increases we saw in L2 cache miss rates for some of the benchmarks are attenuated across the entire application execution time. Since L2 accesses are typically orders of magnitude less than the cycle count for the entire application, modest increases in L2 miss rates should not devastate our runtime performance. Figure 10 shows the impact of the proposed adaptive architecture on the overall execution runtime. As one can see, performance is essentially maintained equivalent to the baseline (100%). Some benchmarks do incur a nominal increase in execution time, but the overhead is typically trivial. For example, the worst-case execution time benchmark (*twolf* using 128KB L2 with global threshold) had a slowdown of 0.05%. It is interesting to note that the four relatively large increases all occur in global threshold cases. Again, when application-specific information is utilized, the general performance of the system improves. As such, we do see an average minor improvement in runtime for the application-specific thresholds, which corresponds as expected with the average miss-rate reductions discussed earlier. In the best case (*parser* using 128 KB with application-specific threshold), the runtime is improved by 0.13%.

With regard to power efficiency, our primary ambition for these mobile smartphones, we were able to observe positive results. Since we only use a nominal amount of additional hardware, the impact of the proposed technique is quite minimal. Overall, the structures proposed account for only 1,428-bytes ($\approx$1.39 KB) of additional storage elements, along with some necessary routing signals and muxing. Furthermore, the

Table IX. Miss-Rate Improvement Using LFSR Instead of MSB

Benchmark	Improvement
bzip2	6.23e-4%
crafty	5.48e-4%
crc32	9.62e-4%
eon	−6.12e-5%
fft	1.42e-5%
gap	3.91e-5%
gcc	−2.27e-5%
gsm	2.01e-4%
gzip	−5.99e-5%
h264dec	2.12e-5%
mcf	1.27e-4%
mpeg4enc	3.36e-5%
parser	1.32e-4%
perlbmk	3.01e-5%
twolf	−7.19e-4%
vortex	2.94e-4%
vpr	1.81e-5%

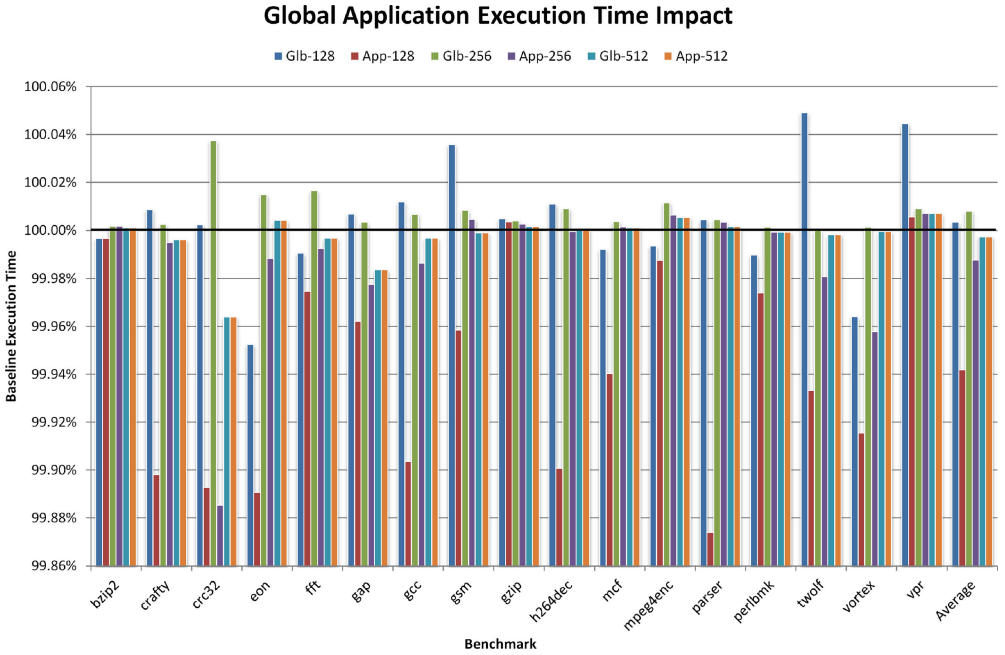

Fig. 10. Global application execution time impact.

overhead of initially downloading and configuring the threshold and decay registers at the start of an application is negligible. The operating system is essentially doing two register writes: one 32-bit write packing the four threshold registers (each 8-bits wide) and one 64-bit write for the $DECR_{reset}$ register. As a hardware register write only take 1–3 processor cycles, the overhead of setting up the registers is greatly outweighed

by the lifetime of the application's execution (being in billions of cycles) and thus is imperceptible in terms of performance or power overhead.

We used CACTI [Wilton and Jouppi 1996] and eCACTI [Mamidipaka and Dutt 2004] to estimate both the dynamic and static power consumption for our caches, assuming a conservative 65nm feature size and leakage temperature of 85°C. Furthermore, we take into account the additional power incurred while doing secondary L2 cache look-ups (on those "expanded" cache sets), as well as the power associated with L2 cache misses and the additional logic proposed by our architecture, including the added metadata storage and access, control logic, and flushing of any dirty cache lines on contraction. Using this information, we are able to determine the approximate total cache subsystem power consumption across the entire runtime for each of the aforementioned benchmarks.

Table X shows the percentage of power improvement attained by our proposed adaptive cache architecture for the complete execution of each benchmark, and a graphical representation is shown in Figure 11. As one can see, we consistently achieve significant reductions in the overall power utilization across all benchmarks. Furthermore, as one would expect, the application-specific results are generally better than our global, across-the-board threshold. On average, our application-aware adaptive cache architecture provides power reductions of 12.96%, 20.25%, and 28.52% for the 128KB, 256KB, and 512KB cache sizes, respectively. Evidently, as the L2 cache size increases, the amount of static power that can be ameliorated by gating idle cache sets becomes more prominent.

It is also interesting to note that there are a few instances where the global threshold outperforms its application-specific equivalent, as can be seen in *crafty*, *gcc*, and *twolf*. In all of these cases, the compiler-generated thresholds for the contraction mechanism (T_{c_off} and T_{c_on}) were less aggressive than those of the hand-selected global threshold. This is caused by the profiled temporal delay mean (TD_{μ}) being pulled lower by a group of cache sets that are very frequently accessed, although there are still many other cache sets that are idle for long periods of time. In all of these particular instances, the temporal delay standard deviation (TD_{σ}) was relatively larger than the other benchmarks, indicating that there were a number of outliers skewing the results. Since our proposal relies on coarse application-wide information to simplify the necessary hardware data structures and complexity of the system, it can be susceptible to such biasing of the overall application memory characteristics. In these rare cases, the benefits are slightly subdued, but the overall power improvement is still much better than the baseline.

As one can see, the trade-off between performance and power in this implementation is quite skewed; we lost only a marginal amount of performance (or in many cases even gained some performance), while gaining substantial overall power reductions. Obviously, there is a limit, where if enough cache sets are gated, the performance will quickly degrade to a point where power consumption is also severely impacted (due to having to access subsequent, larger memory structures and extending the overall runtime). The key is to balance the gating of inactive cache sets while enabling those more highly-used sets to be able to store their information. Based on our results, the compile-time profiling and generation of threshold and decay register values in an application-specific fashion is a success. Using the information gleaned during offline profiling and embedded in the application binary allows our architecture to be more responsive to the nuances of the memory subsystem behavior of a given application.

Finally, to put these results into perspective, a generalized estimate of the cache subsystem's power footprint within the overall processor was performed. Using a substantially modified version of Wattch [Brooks et al. 2000], a power estimation of the experimental processor architecture described earlier was conducted. An instruction

Table X. Total Cache Subsystem Power Improvement

Benchmark	Cache Size	*Global Threshold* Power Improvement			*Application-Specific* Power Improvement		
		Dynamic	Static	Total	Dynamic	Static	Total
bzip2	128 KB	2.53%	29.99%	16.17%	2.22%	28.09%	15.06%
	256 KB	2.36%	31.29%	20.76%	1.96%	32.81%	21.58%
	512 KB	2.80%	35.88%	27.36%	2.34%	36.21%	27.48%
crafty	128 KB	5.78%	33.15%	13.38%	10.59%	15.30%	11.89%
	256 KB	6.99%	39.11%	19.04%	5.62%	21.61%	11.62%
	512 KB	5.18%	35.45%	17.07%	4.06%	38.27%	17.49%
crc32	128 KB	4.17%	21.11%	7.86%	7.52%	21.53%	10.56%
	256 KB	3.69%	31.95%	12.49%	13.05%	39.64%	21.33%
	512 KB	5.67%	24.56%	13.78%	5.27%	42.18%	21.13%
eon	128 KB	3.18%	30.72%	10.89%	6.87%	22.75%	11.31%
	256 KB	−0.23%	34.55%	12.77%	1.16%	40.25%	15.77%
	512 KB	0.27%	31.54%	15.07%	0.47%	48.12%	23.02%
fft	128 KB	1.38%	30.22%	12.08%	5.35%	26.66%	13.26%
	256 KB	−0.70%	33.35%	17.07%	4.84%	42.35%	24.41%
	512 KB	0.69%	34.91%	21.69%	4.38%	46.56%	30.27%
gap	128 KB	−0.31%	31.55%	7.81%	1.95%	25.41%	7.92%
	256 KB	−0.11%	24.46%	9.35%	1.14%	38.09%	15.37%
	512 KB	0.08%	39.49%	18.24%	0.63%	45.10%	21.12%
gcc	128 KB	14.09%	14.06%	14.08%	10.08%	14.45%	11.41%
	256 KB	17.58%	16.00%	16.91%	5.37%	25.43%	13.85%
	512 KB	14.54%	24.69%	19.27%	3.84%	39.51%	20.48%
gsm	128 KB	0.31%	34.55%	7.85%	4.81%	35.81%	11.64%
	256 KB	1.96%	48.09%	17.14%	3.68%	60.17%	22.28%
	512 KB	1.81%	45.10%	19.24%	3.64%	62.10%	27.17%
gzip	128 KB	5.85%	28.15%	16.28%	3.56%	41.44%	21.28%
	256 KB	8.57%	40.19%	28.50%	5.30%	51.39%	34.35%
	512 KB	7.42%	50.54%	36.55%	4.57%	59.09%	41.41%
h264dec	128 KB	3.26%	26.58%	11.55%	11.75%	21.12%	15.08%
	256 KB	4.07%	36.88%	19.22%	4.10%	43.02%	22.06%
	512 KB	4.32%	51.55%	29.91%	3.79%	59.11%	33.76%
mcf	128 KB	1.17%	34.20%	16.15%	7.58%	28.29%	16.97%
	256 KB	−0.20%	46.22%	27.93%	0.12%	47.23%	28.66%
	512 KB	0.09%	51.87%	35.72%	0.18%	52.01%	35.84%
mpeg4enc	128 KB	2.45%	38.90%	16.30%	4.51%	37.21%	16.94%
	256 KB	1.40%	42.32%	22.72%	3.78%	51.06%	28.42%
	512 KB	2.16%	49.88%	31.53%	3.97%	55.56%	35.72%
parser	128 KB	4.42%	36.85%	15.32%	11.96%	21.87%	15.29%
	256 KB	5.34%	35.77%	19.13%	2.70%	49.08%	23.71%
	512 KB	4.91%	49.22%	28.10%	2.56%	60.40%	32.83%
perlbmk	128 KB	1.91%	28.57%	13.96%	7.00%	27.51%	16.28%
	256 KB	0.65%	43.33%	25.58%	4.74%	45.22%	28.38%
	512 KB	0.19%	49.18%	33.09%	4.69%	58.10%	40.56%
twolf	128 KB	−0.36%	34.29%	8.72%	7.22%	14.34%	9.09%
	256 KB	3.56%	40.49%	16.38%	4.80%	23.39%	11.25%
	512 KB	3.48%	55.63%	26.13%	3.99%	49.12%	23.59%

Continued

Table X. Continued

Benchmark	Cache Size	*Global Threshold* Power Improvement			*Application-Specific* Power Improvement		
		Dynamic	Static	Total	Dynamic	Static	Total
vortex	128 KB	2.44%	25.68%	8.89%	5.09%	23.54%	10.21%
	256 KB	0.37%	33.09%	12.80%	2.34%	30.02%	12.85%
	512 KB	0.34%	48.48%	23.24%	0.32%	56.04%	26.83%
vpr	128 KB	2.38%	14.56%	5.62%	2.35%	16.21%	6.03%
	256 KB	5.20%	13.59%	8.31%	2.80%	17.93%	8.40%
	512 KB	5.40%	41.69%	22.24%	2.87%	53.02%	26.15%
Average	128 KB	3.21%	29.01%	11.93%	6.50%	24.80%	12.96%
	256 KB	3.56%	34.75%	18.01%	3.97%	38.75%	20.25%
	512 KB	3.49%	42.33%	24.60%	3.03%	50.62%	28.52%

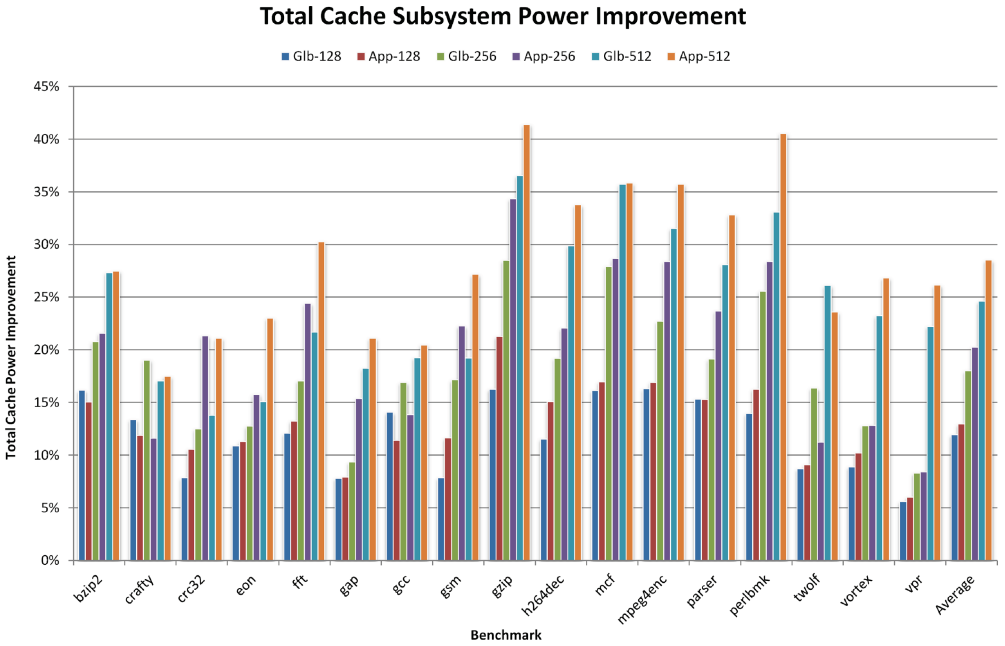

Fig. 11. Total cache subsystem power improvement.

breakdown of 50% integer, 5% float point, 20% load, 15% store, and 10% control was assumed. Cache miss rates were also assumed to be 2%, 4%, and 10% for the L1 instruction cache, L1 data cache, and unified L2 cache, respectively. The percentage of the overall processor power due to the cache subsystem was estimated to be 31.91%, 43.65%, and 51.28% for the 128KB, 256KB, and 512KB L2 cache configurations, respectively.

7. TIMING CONSIDERATIONS

An important aspect of cache architecture design is the timing delay involved with cache hits. The cache subsystem is typically on the critical path, and there is often very little timing slack to allow for additional complex logic calculations in the sequential access path. For example, it would be impractical to do two sequential cache lookups in a single cycle. Instead, to achieve the behavior described earlier for the expansion

mechanism, the pseudo-associative secondary lookup would require an additional cycle. In essence, this secondary access behaves like an initial cache miss, stalling the pipeline for one cycle to reaccess the cache again in another location.

The updating of a cache set's frequency shift register (FSR), expand bit, and MRU/gate bit all occur in parallel to a cache access and do not impact the critical path. These modifications will be completed within the current access cycle, and will then impact cache behavior on future accesses to that given cache set.

The only impact to the critical path of our cache design is the accessing of the MRU/gate bit to determine whether to access the primary or secondary cache location. Given the MSB toggling mapping function we utilized, the overhead is effectively just an additional one-bit *XOR* gate on the most-significant bit of the cache index, mollifying any concerns of possible timing violations.

8. CONCLUSIONS

As shown, caches contribute a significant amount of the overall processor power budget, both in terms of dynamic and leakage power. Although much work has gone into mitigating cache power consumption, mobile processors still suffer from large caches that are necessary to bridge the growing memory latency gap. Mobile processors, being far more constrained in terms of power consumption and area constraints, embody a unique usage model that demands continuous access while having a limited battery life. Ultra-low-power architectural solutions are required to help meet these consumer demands.

We have presented a novel architecture for leveraging compile-time profiling information and significantly reducing overall cache power consumption in high-performance mobile processors. The key principle is the strengthening of the interaction between the compiler and the underlying hardware microarchitecture, allowing information not readily available during runtime (e.g., global average miss rates, access rates, and temporal access delays) to be gathered and conveyed to the hardware by the compiler. By leveraging application-specific information and tuning the cache to adaptively respond to runtime memory behavior, a significant decrease in power usage can be achieved with little sacrifice in terms of speed or area. This has been demonstrated by using a representative set of simulation benchmarks. The proposed technique has significant implications for mobile processors, especially high-performance, power-sensitive devices, such as smartphones, as it significantly reduces power consumption with minimal degradation in runtime performance. End users can still enjoy their high-performance mobile device while also enjoying longer battery life.

REFERENCES

AGARWAL, A., LI, H., AND ROY, K. 2002. DRG-cache: A data retention gated-ground cache for low power. In *Proceedings of the 39th Conference on Design Automation (DAC'02)*. 473–478.

AGARWAL, A. AND PUDAR, S. D. 1993. Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. *SIGARCH Comput. Architect. News, 21*, 2, 179–190.

ALBONESI, D. H. 1999. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the 32nd International Symposium on Microarchitecture (MICR0'32)*. 248–259.

AUSTIN, T., LARSON, E., AND ERNST, D. 2002. SimpleScalar: An infrastructure for computer system modeling. *Computer 35*, 2, 59–67.

BOURNOUTIAN, G. AND ORAILOGLU, A. 2008. Miss reduction in embedded processors through dynamic, power-friendly cache design. In *Proceedings of the 45th Conference on Design Automation (DAC'08)*. 304–309.

BOURNOUTIAN, G. AND ORAILOGLU, A. 2010. Dynamic, non-linear cache architecture for power-sensitive mobile processors. In *Proceedings of the 8th International Conference on Hardware/Software Codesign and System Synthesis (CODES/ISSS'10)*. 187–194.

Brooks, D., Tiwari, V., and Martonosi, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA'00)*. 83–94.

Calder, B., Grunwald, D., and Emer, J. 1996. Predictive sequential associative cache. In *Proceedings of the 2nd Symposium on High-Performance Computer Architecture (HPCA'96)*. 244–253.

Flautner, K., Kim, N. S., Martin, S., Blaauw, D., and Mudge, T. 2002. Drowsy caches: Simple techniques for reducing leakage power. *SIGARCH Comput. Architect. News 30,* 2, 148–157.

Ghose, K. and Kamble, M. B. 1999. Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'99)*. 70–75.

Gordon-Ross, A., Vahid, F., and Dutt, N. 2009. Fast configurable-cache tuning with a unified second-level cache. *IEEE Trans. Very Large Scale Integ. Syst. 17,* 1, 80–91.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the International Workshop on Workload Characterization (WWC'01)*. 3–14.

Hasegawa, A., Kawasaki, I., Yamada, K., Yoshioka, S., Kawasaki, S., and Biswas, P. 1995. SH3: High code density, low power. *IEEE Micro 15,* 6, 11–19.

Inoue, K., Ishihara, T., and Murakami, K. 1999. Way-predicting set-associative cache for high performance and low energy consumption. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'99)*. 273–275.

ITRS. 2009. Semiconductor Industry Association. International Technology Roadmap for Semiconductors, 2009. http://www.itrs.net/.

Jouppi, N. P. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *SIGARCH Comput. Architect. News, 18*, 2SI, 364–373.

Kin, J., Gupta, M., and Mangione-Smith, W. H. 1997. The filter cache: An energy efficient memory structure, In *Proceedings of the 30th Intenational Symposium on Microarchitecture (MICRO'30)*. 184–193.

Ko, U., Balsara, P. T., and Nanda, A. K. 1995. Energy optimization of multi-level processor cache architectures. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'95)*. 45–49.

Lee, C., Potkonjak, M., and Mangione-Smith, W. H. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th International symposium on Microarchitecture (MICRO30)*. 330–335.

Lee, L., Kannan, S., and Fridman, J. 2004. MPEG4 video codec on a wireless handset baseband system. In *Proceedings of the Workshop Media and Signal Processors for Embedded Systems and SoCs*.

Malik, A., Moyer, B., and Cermak, D. 2000. A low power unified cache architecture providing power and performance flexibility. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'00)*. 241–243.

Mamidipaka, M. and Dutt, N. 2004. eCACTI: An enhanced power estimation model for on-chip caches. University of California, Irvine Center for Embedded Computer Systems. Tech. rep. TR-04-28.

Nethercote, N. and Seward, J. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'07)*. 89–100.

Powell, M., Yang, S.-H., Falsafi, B., Roy, K., and Vijaykumar, T. N. 2000. Gated-V_{dd}: A circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'00)*. 90–95.

Rodriguez, S. and Jacob, B. 2006. Energy/power breakdown of pipelined nanometer caches (90nm/65nm/45nm/32nm). In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'06)*. 25–30.

SPEC. 2000. SPEC CPU2000 Benchmarks. http://www.spec.org/cpu/.

Sundararajan, K. T., Jones, T. M., and Topham, N. 2011. Smart cache: A self adaptive cache architecture for energy efficiency. In *Proceedings of the International Conference on Embedded Computer Systems (SAMOS'11)*. 41–50.

Wilton, S. J. E. and Jouppi, N. P. 1996. CACTI: An enhanced cache access and cycle time model. *IEEE J. Solid-State Circuits 31,* 5, 677–688.

Received March 2011; revised October 2011, April, October 2012; accepted December 2012