# Dynamic, Non-Linear Cache Architecture for Power-Sensitive Mobile Processors


Garo Bournoutian
University of California, San Diego
9500 Gilman Dr. #0404
La Jolla, CA 92093-0404
garo@cs.ucsd.edu

Alex Orailoglu
University of California, San Diego
9500 Gilman Dr. #0404
La Jolla, CA 92093-0404
alex@cs.ucsd.edu



## ABSTRACT

*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 paper explores a novel approach to mitigating mobile processor power consumption, with a non-linear degradation in execution speed. The concept relies on using dynamic 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 16% to 19%, 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/mobile architecture*;
C.3 [**Special-Purpose and Application-Based Systems**]: real-time and embedded systems

## General Terms

Design, Performance

## Keywords

mobile processors, low-power cache design, dynamic, power-sensitive



Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*CODES+ISSS'10,* October 24–29, 2010, Scottsdale, Arizona, USA.
Copyright 2010 ACM 978-1-60558-905-3/10/10 ...$10.00.


## 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. Even today, 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 multi-level data caches, complex branch prediction, and now even multi-core architectures, such as the Qualcomm Snapdragon and ARM Cortex-A9.

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, e-mails, internet browsing, news, music, video, TV, and games. Furthermore, these devices need to be ultra-portable, being carried unobtrusively on a person and requiring extremely infrequent power access to recharge. In addition, due to their small form factor, these devices often have reduced storage capacity and instead rely on remote data streaming.

With the constraints embodied by mobile processors, one typically is concerned with high performance, power efficiency, better execution determinism, and minimized area. Unfortunately, these characteristics are often adversarial, and focusing on improving one often results in worsening the others. For example, in order to increase performance, one adds a more complex cache hierarchy 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 [1].

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 so-

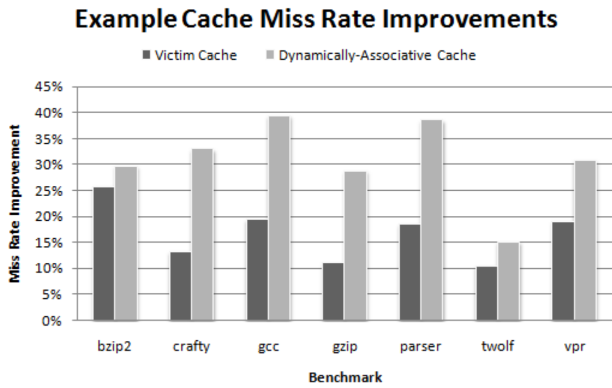


**Figure 1: Example of Cache Thrashing Reductions**

lutions such as the victim cache [2] or dynamically-associative caches [3] to improve cache hit rates, as shown in Figure 1. 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 in order to satisfy the desired longevity in these devices. Furthermore, static leakage power is becoming increasingly more predominant as feature size continues to be reduced, and can be as much as 30-40% of the total power consumed by the processor [4]. Since caches typically account for 30-60% of the total processor area and 20-50% of the processor's power consumption, they are an ideal candidate for improvement to help reduce overall processor power.

In this paper, 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 small amount of that execution speed in order to prolong the overall life of the mobile device. The key to our approach is that this sacrifice of speed be non-linearly related to the amount of power saved, allowing large gains in 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. By responding dynamically to the application's run-time 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 show the 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 significant improvement in overall processor power consumption of approximately 16% to 19%, 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 multi-player 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. Unfortunately, embedded mobile processors are often more highly constrained than general-purpose processors, and require extra care to minimize power consumption in order to extend device life.

Common structural techniques rely on segmenting the word- or bit-lines in order to reduce latency and dynamic power. Subbanking [5] divides the data arrays into smaller sub-groups, and only the "bank" that contains the desired data is accessed, avoiding wasted bit-line pre-charging dissipation. The Multiple-Divided Module (MDM) Cache [6] 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 [7] first access the tag and then the data arrays. Only on a hit is the data way accessed, resulting in less data way 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 [8] 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 [9], which dynamically shut down parts of the cache according to application demand, suffer from sharp performance degradation when aggressively applied. Similarly, Speculative Way Activation [10, 11] 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. Unfortunately, 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.

There are also several techniques that specifically attempt to target leakage power within caches. The Gated-$V_{dd}$ Technique [12] 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 [13] reduces the leakage power significantly, while still retaining the data within the memory cells while in standby mode. Unfortunately, there is also a significant increase to the word-line delay of 60% or higher depending on the feature-size. Similarly, the Drowsy Cache [14] provides a low-power "drowsy"

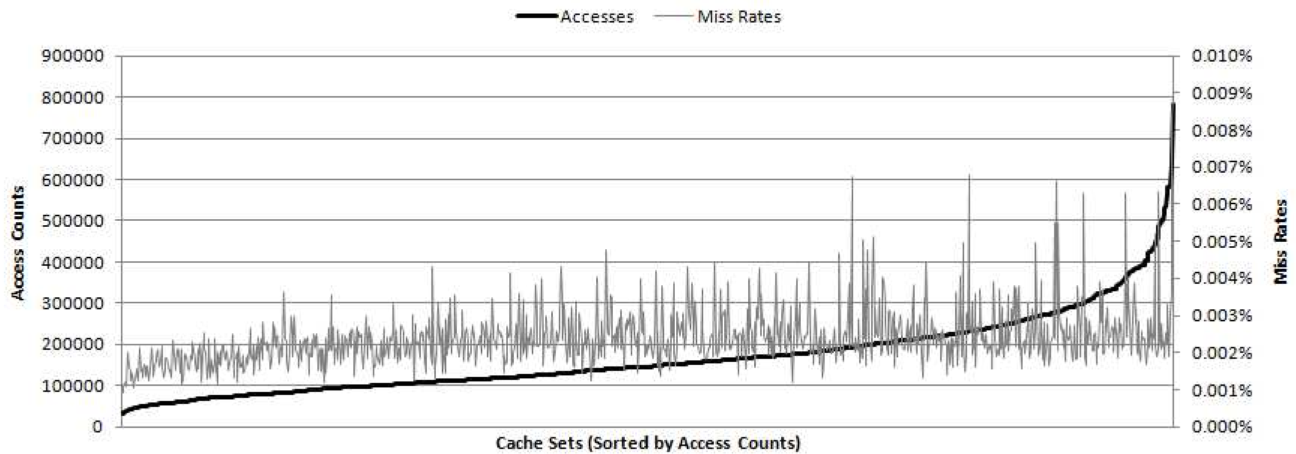


**Figure 2: L2 Cache Access and Miss Rate Distibution for *gcc***

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 run-time 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 hot-spots, but if they happen to exist within the same hot-spot, each time one is brought into the cache, the other will be evicted, and this *thrashing* will continue for the entire hot-spot.

Given complex and data-intensive applications, the probability of multiple cache lines being active within a hot-spot, 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) and tertiary (L3) caches contribute the 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 and L3 caches exhibit 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 and L3 caches. Because of this observation, much of the L2 and L3 caches are idle for extended periods of time. We can take advantage of this inactivity and temporarily shut down those idle cache sets in order 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 2 provides a sorted distribution of L2 cache accesses per cache set for the *gcc* 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. Furthermore, one can observe that few peaks within the miss rate series that exceed 0.005% occur in the most actively accessed portions of the cache. These peaks are most likely caused by conflicts and thrashing.

The goal of this paper 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.

# 4. IMPLEMENTATION

The proposed solution enables two complementary types of behavior to occur per cache set: *expansion* and *contrac-*

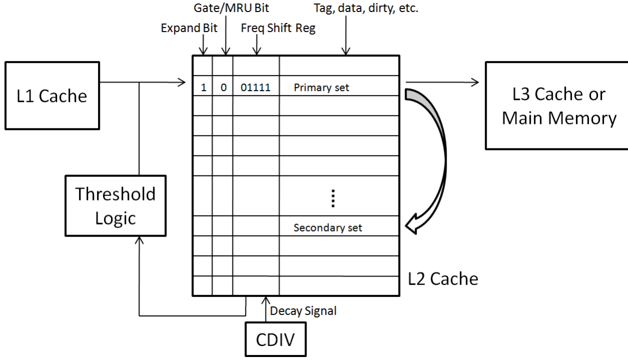


**Figure 3: High-Level Cache Implementation**

*tion*. Each cache set will be annotated with a small number of additional bits to keep track of the new state information. 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 clock divider, 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 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:

$$T_{e_on} > T_{e_off} \geq T_{c_off} > T_{c_on}$$

Thus, the minimum size of the threshold registers (and also the *FSR*) is 3-bits, and we will denote this size with $N$.

To implement this dynamic, non-linear 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). 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 8-bit *FSR* ($N = 8$) and have 1024 sets in our implementation, this results in an additional 1412-bytes (≈1.38KB) added to our L2 cache; evidently, a negligible amount of storage compared to the actual cache size of 256KB.

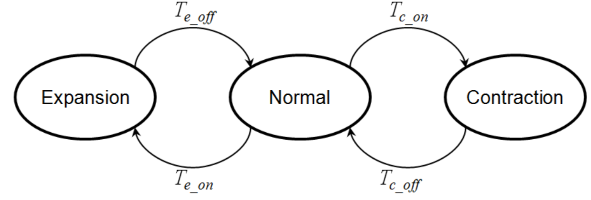


**Figure 4: Cache Behavior State Diagram**

The details of both the expansion and contraction mechanisms are described in detail in the following sections.

## 4.1 Expansion Mechanism

This mechanism allows for the expansion of a particular cache set into a second cache set, similar to the architecture described in [3]. 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 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 to decompose the cache into two disjoint maximal loops[1]. 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.

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. We need not worry that by using the complement set we may pollute a currently active set in our cache. 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.

[1]This allowed us to avoid the all-zero state within our LFSR which constitutes one of the maximal loops.

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*.

## 4.2 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 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 [12]. As shown in Figure 5, 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.

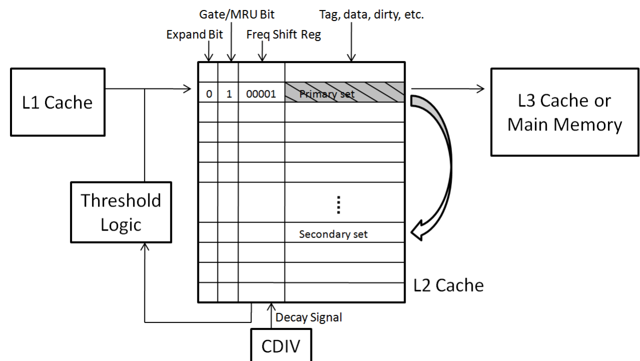


**Figure 5: Example Gated Primary Set Redirecting Queries to Secondary Location**

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.

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 and L3 caches 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.

# 5. EXPERIMENTAL RESULTS

In order to assess the benefit from this proposed architectural design, we utilized the SimpleScalar toolset [15]. We chose a representative mobile smartphone system configuration, having 32KB L1 data and instruction caches (1024-set, direct-mapped with a 32-byte line size), and a 256KB L2 cache (1024-set, 4-way set-associative with a 64-byte line size). Additionally, we chose an 8-bit *Frequency Shift Register* (*FSR*). This processor model utilized a typical in-order simulation engine, with dedicated L1 caches for data and instruction memory, backed by a unified L2 cache.

In addition to the baseline, non-modified cache architecture, we provide results for three different configuration levels, each increasingly more aggressive with respect to power conservation: *Config 0*, *Config 1*, *Config 2*. The associated threshold values for each configuration are available below in Table 1.

| Threshold | *Config 0* | *Config 1* | *Config 2* |
|---|---|---|---|
| $T_{e_on}$ | 00010000 | 01000000 | 10000000 |
| $T_{e_off}$ | 00001000 | 00010000 | 01000000 |
| $T_{c_off}$ | 00000010 | 00001000 | 00010000 |
| $T_{c_on}$ | 00000001 | 00000010 | 00000100 |

**Table 1: Threshold Values for Experimental Configurations**

| | Baseline | | *Config 0* | | *Config 1* | | *Config 2* | |
|---|---|---|---|---|---|---|---|---|
| Benchmark | Accesses | Miss Rate | Miss Rate | Improvement | Miss Rate | Improvement | Miss Rate | Improvement |
| bzip2 | 4.29e9 | 0.1699 | 0.1393 | 18.01% | 0.1698 | 0.06% | 0.1710 | -0.65% |
| crafty | 3.42e9 | 0.0038 | 0.0034 | 10.53% | 0.0039 | -2.63% | 0.0040 | -5.26% |
| crc32 | 4.26e7 | 0.1750 | 0.1473 | 15.83% | 0.1752 | -0.11% | 0.1797 | -2.69% |
| fft | 2.48e7 | 0.1007 | 0.0891 | 11.52% | 0.1013 | -0.60% | 0.1049 | -4.17% |
| gcc | 1.72e8 | 0.0244 | 0.0197 | 19.26% | 0.0249 | -2.05% | 0.0252 | -3.28% |
| gsm | 6.00e7 | 0.1222 | 0.1015 | 16.94% | 0.1225 | -0.25% | 0.1231 | -0.74% |
| gzip | 3.65e9 | 0.0147 | 0.0126 | 14.29% | 0.0148 | -0.68% | 0.0150 | -2.04% |
| h264dec | 6.69e8 | 0.0973 | 0.0878 | 9.76% | 0.0977 | -0.41% | 0.0991 | -1.85% |
| mpeg4enc | 4.24e8 | 0.0917 | 0.0798 | 12.98% | 0.0924 | -0.76% | 0.0946 | -3.16% |
| parser | 7.65e8 | 0.0814 | 0.0694 | 14.74% | 0.0817 | -0.37% | 0.0822 | -0.98% |
| twolf | 1.10e9 | 0.1148 | 0.1065 | 7.23% | 0.1147 | 0.09% | 0.1148 | -0.01% |
| vpr | 6.92e8 | 0.1425 | 0.1177 | 17.40% | 0.1425 | 0.01% | 0.1439 | -0.98% |

**Table 2: L2 Cache Miss Rates**

Twelve representative benchmark applications from the SPEC CPU2000 suite [16], the MediaBench video suite [17], and the MiBench telecomm suite [18] are used: *bzip2* - compression program; *crafty* - high-performance chess playing application; *crc32* - 32-bit CRC framing checksum; *fft* - discrete fast Fourier transform program; *gcc* - C compiler; *gsm* - telecomm speech transcoder compression program; *gzip* - LZ77 compression program; *h264dec* - H.264 video decoder; *mpeg4enc* - MPEG-4 video encoder; *parser* - word processing application; *twolf* - CAD place-and-route simulation; and *vpr* - FPGA place-and-routing.

Table 2 provides the miss rate comparison among the various configurations using the MSB mapping function for the *Expansion* and *Contraction* mechanisms. As one can see, *Config 0*, which is least aggressive in terms of power reduction and instead is inclined to enable *expansion* more often, results in notable improvements to L2 miss rate. Thus, were power conservation not a critical issue, one could run in *Config 0* and achieve a performance increase. On the other hand, *Config 2*, which is most aggressive in terms of power reduction, does result in a tangible increase to miss rates. Figure 6 graphically shows the miss rate comparisons across the benchmarks and various configurations. The average miss rates for the baseline, *Config 0*, *Config 1*, and *Config 2* are 9.49%, 8.12%, 9.51%, and 9.64%, respectively. As one can see, *Config 1* and *Config 2* do result in slightly larger miss rates of approximately 0.2% and 1.6%, respectively, relative to the baseline.

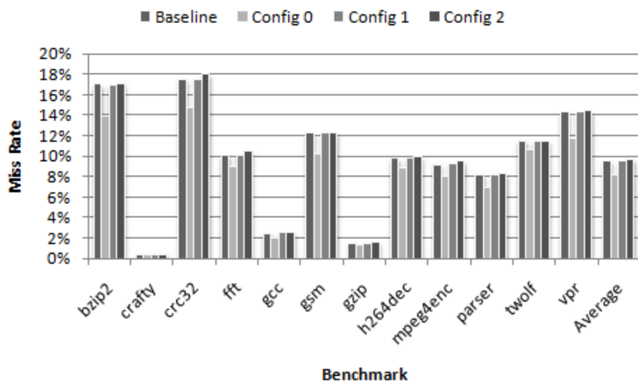


**Figure 6: L2 Cache Miss Rates**

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 3 provides miss rate improvements over MSB when using an LFSR with a cycle length of 512 (i.e. two disjoint loops within our 1024-set cache). As one can see, the improvements are extremely miniscule, being less than a thousandth of a single percent, and in some instances even worse than the MSB approach. Given this and the associated overhead in hardware complexity and area to implement an LFSR mapping function, the MSB approach appears to be more efficient. Thus, we use the MSB mapping function for the remainder of our calculations.

Luckily, the increases we saw in L2 cache miss rates for the more aggressive configurations are attenuated across the

| Benchmark | *Config 0* | *Config 1* | *Config 2* |
|---|---|---|---|
| bzip2 | 7.18e-4% | 6.48e-4% | 1.33e-5% |
| crafty | 5.88e-4% | 2.56e-4% | 5.76e-5% |
| crc32 | 1.36e-3% | 2.85e-4% | 6.14e-4% |
| fft | 1.13e-5% | 3.95e-4% | 3.84e-5% |
| gcc | -1.02e-5% | -2.71e-5% | 1.29e-5% |
| gsm | 5.91e-4% | 8.01e-4% | 9.67e-5% |
| gzip | -9.29e-5% | 2.05e-4% | -1.99e-4% |
| h264dec | 7.43e-5% | 9.92e-5% | 4.97e-5% |
| mpeg4enc | 4.49e-5% | 7.23e-4% | 3.11e-4% |
| parser | 1.09e-4% | 1.87e-4% | 9.75e-5% |
| twolf | -6.16e-4% | 7.73e-5% | 4.71e-5% |
| vpr | 6.63e-5% | 5.91e-5% | -7.38e-4% |

**Table 3: Miss Rate Improvement Using LFSR Instead of MSB**

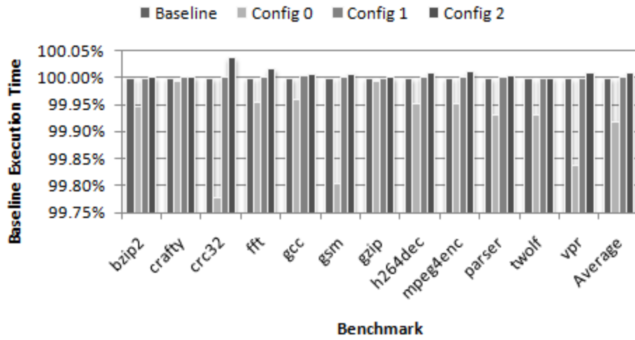


**Figure 7: Global Application Execution Time Impact**

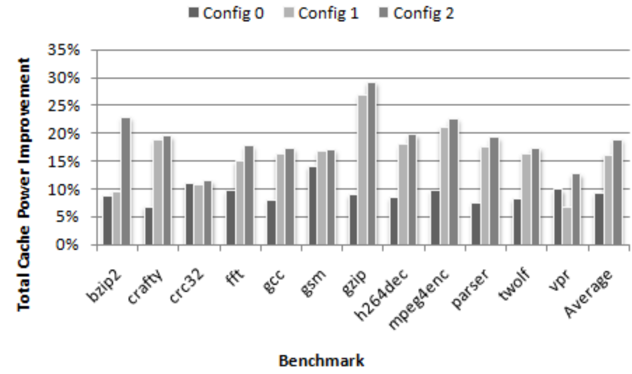


**Figure 8: Total Cache Power Improvement**

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 run-time performance. Figure 7 shows the impact of the various configurations on the overall execution run-time. As one can see, the more performance-centric *Config 0* is able to consistently reduce execution time. On the other hand, the more power-conscientious configurations (*Config 1* and *Config 2*) do have slight increases in execution time. Yet, even the worst benchmark (*crc32*) had a run-time increase of less than 0.05%, and the average across all benchmarks was around 0.01%.

With regard to power efficiency, our primary ambition for these mobile smartphones, we were able to observe excellent 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 1412-bytes (≈1.38KB) of additional storage elements, along with some necessary routing signals and muxing.

We used CACTI [19] and eCACTI [20] to estimate both the dynamic and static power consumption for our caches, assuming a 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. Using this information, we are able to determine the approximate total cache subsystem power consumption across the entire run-time for each of the aforementioned benchmarks.

| Benchmark | *Config 0* | *Config 1* | *Config 2* |
|---|---|---|---|
| bzip2 | 8.65% | 9.53% | 22.82% |
| crafty | 6.78% | 18.75% | 19.46% |
| crc32 | 10.92% | 10.58% | 11.40% |
| fft | 9.60% | 14.98% | 17.66% |
| gcc | 7.95% | 16.18% | 17.25% |
| gsm | 14.05% | 16.69% | 17.01% |
| gzip | 8.94% | 26.90% | 29.16% |
| h264dec | 8.40% | 17.88% | 19.71% |
| mpeg4enc | 9.67% | 21.08% | 22.60% |
| parser | 7.49% | 17.59% | 19.19% |
| twolf | 8.18% | 16.29% | 17.31% |
| vpr | 9.81% | 6.76% | 12.64% |

**Table 4: Total Cache Power Improvement**

Table 4 shows the percentage of power improvement attained by the three configurations for the complete execution of each benchmark, and a graphical representation is shown in Figure 8. As one can see, we consistently achieve significant reductions in the overall power utilization across all benchmarks. On average, *Config 0*, *Config 1*, and *Config 2* provide reductions of 9.21%, 16.10%, and 18.85%, respectively. The highest reduction in power occurred in the *gzip* benchmark, with *Config 2* (our more aggressive power configuration) garnering a 29.16% decrease!

As one can see, there is definitely a non-linear trade-off to the various levels of aggressiveness delineated by our three configurations. With each successively more zealous configuration, we lost only a marginal amount of 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 run-time). 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.

# 6. 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, in order 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 re-access 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.

# 7. FUTURE EXTENSIONS

While the architecture presented in this paper assumed fixed, pre-defined threshold values, it should be noted that an application-specific approach can also be taken. One could have the compiler statically analyze a given application and determine the ideal (or near-ideal) threshold values for that particular application. The compiler can then embed this information within the application binary, and upon the operating system loading that application into the processor, the *loader* can convey these values to the underlying microarchitecture. In this manner, the threshold values do not need to remain static for the processor, but rather can dynamically change on a per-application basis. This would enable a more fine-grained capability for matching the cache behavior with a given application, and may yield even further power and performance benefits.

# 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 cellular 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 significantly reducing overall cache power consumption in high-performance mobile processors. The achievement of these goals has been confirmed by extensive experimental results. By using varying configurations, a notable increase in cache hit rates can be achieved when plugged-in or fully-charged, while a significant decrease in power usage can be achieved when the user switches the device into low-power mode. This has been demonstrated by using a representative set of simulation benchmarks, and three example configurations. The proposed technique has significant implications for mobile processors, especially high-performance, power-sensitive devices such as smartphones, as it significantly reduces power consumption while minimally degrading run-time performance.

As embedded mobile processors continue to spread and become ubiquitous, it is essential to maintain high performance, low power, and small size. The proposed architecture fulfills these requirements and enables mobile processors to continue to mature and be able to handle exceedingly complex and aggressive applications.

# 9. REFERENCES


[1] Li Lee, Srikanth Kannan, and Jose Fridman. MPEG4 video codec on a wireless handset baseband system. In *Proc. Workshop Media and Signal Processors for Embedded Systems and SoCs*, 2004.

[2] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *SIGARCH Computer Architecture News*, pages 364–373, 1990.

[3] Garo Bournoutian and Alex Orailoglu. Miss reduction in embedded processors through dynamic, power-friendly cache design. In *DAC '08: Proc. 45th Annual Conference on Design Automation*, pages 304–309, New York, NY, 2008.

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

[5] Kanad Ghose and Milind B. Kamble. Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In *ISLPED '99: Proc. 1999 International Symposium on Low Power Electronics and Design*, pages 70–75, New York, NY, 1999.

[6] Uming Ko, Poras T. Balsara, and Ashwini K. Nanda. Energy optimization of multi-level processor cache architectures. In *ISLPED '95: Proc. 1995 International Symposium on Low Power Electronics and Design*, pages 45–49, New York, NY, 1995.

[7] Atsushi Hasegawa, Ikuya Kawasaki, Kouji Yamada, Shinichi Yoshioka, Shumpei Kawasaki, and Prasenjit Biswas. SH3: High code density, low power. *IEEE Micro*, 15(6):11–19, 1995.

[8] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. The filter cache: an energy efficient memory structure. In *MICRO 30: Proc. 30th Annual International Symposium on Microarchitecture*, pages 184–193, Washington, DC, 1997.

[9] David H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *MICRO 32: Proc. 32nd Annual International Symposium on Microarchitecture*, pages 248–259, Washington, DC, 1999.

[10] Brad Calder, Dirk Grunwald, and Joel Emer. Predictive sequential associative cache. In *HPCA '96: Proc. 2nd Symposium on High-Performance Computer Architecture*, pages 244–253, Washington, DC, 1996.

[11] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *ISLPED '99: Proc. 1999 International Symposium on Low Power Electronics and Design*, pages 273–275, New York, NY, 1999.

[12] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. Gated-$V_{dd}$: a circuit technique to reduce leakage in deep-submicron cache memories. In *ISLPED '00: Proc. 2000 International Symposium on Low Power Electronics and Design*, pages 90–95, New York, NY, 2000.

[13] Amit Agarwal, Hai Li, and Kaushik Roy. DRG-cache: a data retention gated-ground cache for low power. In *DAC '02: Proc. 39th Annual Conference on Design Automation*, pages 473–478, New York, NY, 2002.

[14] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: simple techniques for reducing leakage power. *SIGARCH Computer Architecture News*, 30(2):148–157, 2002.

[15] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, pages 59–67, 2002.

[16] SPEC CPU2000 Benchmarks. http://www.spec.org/cpu/.

[17] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO 30: Proc. 30th Annual International Symposium on Microarchitecture*, pages 330–335, Washington, DC, 1997.

[18] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proc. International Workshop on Workload Characterization*, pages 3–14, Washington, DC, 2001.

[19] Steven J. E. Wilton and Norman P. Jouppi. CACTI: An enhanced cache access and cycle time model. *IEEE Journal on Solid-State Circuits*, 31(5):677–688, 1996.

[20] Mahesh Mamidipaka and Nikil Dutt. eCACTI: An enhanced power estimation model for on-chip caches. *University of California, Irvine Center for Embedded Computer Systems Technical Report TR-04-28*, 2004.