Dynamic, Multi-Core Cache Coherence 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, memory-intensive applications that were originally designed and coded for general-purpose processors. However, these mobile processors are also expected to be compact, ultra-portable, and provide an always-on, continuous data access paradigm necessitating a low-power design. As mobile processors increasingly begin to leverage multi-core functionality, the power consumption incurred from maintaining coherence between local caches due to bus snooping becomes more prevalent. This paper explores a novel approach to mitigating multi-core processor power consumption in mobile smartphones. By using dynamic application memory behavior, one can intelligently target adjustments in the cache coherency protocol to help reduce the overhead of maintaining consistency when the benefits of multi-core shared cache coherence are muted. On the other hand, by utilizing a fine-grained approach, the proposed architecture can still respond to and enable the benefits of hardware cache coherence in situations where the performance improvements greatly outweigh the associated energy costs. The simulation results show appreciable reductions in overall cache power consumption, with negligible impact to overall execution time.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Cellular/mobile architecture*; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*

General Terms

Design, Performance

Keywords

mobile processors, multi-core, low-power, cache coherence, dynamic, power-sensitive

CODES+ISSS'11, October 9–14, 2011, Taipei, Taiwan.

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 more recently, 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 the capability to continuously access data 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 improving one often results in worsening the others. For example, in order to increase performance, a more complex cache hierarchy is used 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 excessive cache misses. For example, video codecs, which are increasingly included in wireless devices such as mobile phones, utilize large data footprints and significantly suffer from cache thrashing [1].

In particular, in mobile phone systems, where power and area efficiency are paramount, smaller, less-associative caches

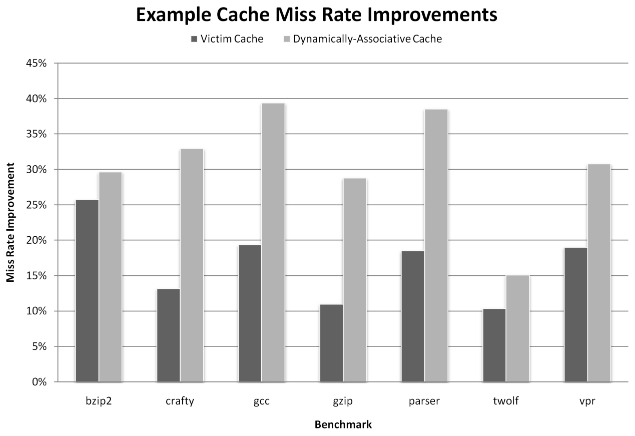


Figure 1: Example of Cache Thrashing Reductions

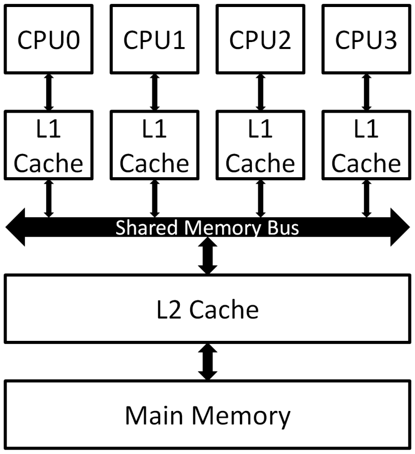


Figure 2: Quad-Core Cache Hierarchy

are typically chosen. Earlier researchers realized that these caches are more predisposed to thrashing and proposed solutions such as a victim cache [2] or dynamically-associative cache [3] to improve cache hit rates, as shown in Figure 1. While these approaches can help mitigate cache thrashing and result in improved execution speed and associated power reduction by avoiding lower level memory subsystem accesses, the shift to multi-core systems introduces new aspects in terms of power utilization.

Shared-memory Multi-Core System-on-Chip (MCSoC) systems are becoming prevalent in the mobile smartphone market, as they provide benefits such as low communication latency, a well-understood programming model, and a decentralized topology well-suited for heterogeneous processing cores. Since all the cores typically share a common bus for their memory accesses, the available bandwidth can become overwhelmed and cause performance degradation. To alleviate this problem, L1 caches are typically localized within each core, saving bus bandwidth and minimizing memory contention. An example of a typical quad-core cache architecture is shown in Figure 2.

This replication of L1 caches introduces the possibility of data corruption if data is being shared among cores, since modifications of locally-stored data in one core may leave other cores with stale versions in their caches which should no longer be considered valid. To rectify this issue, bus-snooping cache coherency protocols are typically employed since they are well suited for embedded MCSoC platforms using a shared high-speed memory bus. The broadcast nature of the memory requests on the shared bus enables all local cache controllers to *snoop* memory transactions from the other cores. Invalidation and write-back signals are used to ensure coherence between all local caches and the lower memory hierarchy (L2 cache in this case). Unfortunately, as we encounter increased cache accesses and related cache misses due to the running of more complex and irregular application sets, the system's cache coherence overhead also increases.

Prior research has shown that bus-snooping cache lookups can amount to 40% of the total power consumed by the cache subsystem in a multi-core processor [4]. Since caches typically account for 30-60% of the total processor area and 20-50% of the processor's power consumption, the overhead from cache snooping is rather significant across the system as a whole. Therefore, the additional power consumption inherent in bus-snooping is often prohibitive and is typically avoided in most mobile processors to date. Instead, these power-sensitive mobile processors rely on purely write-through local caches or software-based coherence. Unfortunately, these approaches have their own drawbacks. Leveraging write-through local caches increases the bus contention and dynamic power when large quantities of data are modified, since every L1 cache modification requires updating the L2 cache as well. Software-based coherence provides coarse-grain enforcement, and the efficiency varies greatly based on the software's coding. Furthermore, to function accurately, it needs to predict four conditions: (i) whether two memory references are to the same location; (ii) whether two memory references are executed on different cores; (iii) whether a conditional write will actually be executed; and (iv) when a write will be executed relative to a sequence of reads. Given the increased complexity and variability of applications that are expected to run on these devices, factors including conditional branches, pointers, dynamic memory, interrupts, and OS context switches will all exacerbate the effectiveness of a static approach. As a result, software approaches will insert invalidation instructions at any point where there may be a possible need. This can lead to numerous unnecessary cache misses and instruction count increases. In contrast, by using a hardware-based approach, one can respond in a fine-grained manner to any given application's run-time behavior.

In this paper, we propose a novel approach in dealing with the unique constraints of mobile processors while enabling a dynamic, power-efficient hardware cache coherency protocol. While the processor normally operates in a baseline configuration, wherein the battery life may be muted in order to deliver full execution speed and responsiveness, a user may instead 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 negligible at first compared to the amount of power saved, allowing a continuum of large gains in power savings without significantly disrupting execution performance. When we need to go into a battery-saving mode, the cache coherence can be dynamically adjusted to reduce power while minimally impacting performance. We show the implementation of this architecture and provide experimental data taken over a general sample of complex, real-world multi-core applications to show the benefits of such an approach. The simulation results show significant improvement in overall cache subsystem power consumption of approximately 15%, while incurring a negligible 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 challenges of memory access. To exacerbate the situation, the mobile processor domain is beginning to leverage multi-core architectures, which then leads to overheads involving cache coherence. 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.

A proposal using virtually tagged caches was presented in [4], where much of the TLB overhead from coherence could be avoided by using virtual addresses in the tag instead of physical addresses. Unfortunately, most systems will opt to have virtually indexed, physically tagged primary caches to avoid complexities arising from aliasing and homonyms.

Serial snooping techniques have been proposed to alleviate overhead from read misses, wherein the assumption of data locality among various processor cores is exploited [5]. In general, these approaches aim to reduce the overhead of snooping all other cores in parallel, but rather do it in a serial fashion starting with the closest cores and move outward. One important shortcoming of such an approach is that latency can increase as we continue to conduct the snooping iteratively, unless the data is residing in the closest neighboring cores. For example, the authors in [5] reported an average increase in latency of 6.25%. Given the importance of low memory latency and its impact on execution speed for mobile smartphones, this approach is often unacceptable.

The *Jetty* approach essentially adds a small directory-like structure between the shared memory structure and each replicated cache [6]. This structure will indicate whether a copy may exist within a particular local cache, helping to avoid tag accesses if the item is guaranteed not to exist in the cache. *Jetty* was proposed for a system where both L1 and L2 were localized, wherein the cost associated with L2 lookups was far greater than the smaller *Jetty* structure. Unfortunately, most mobile processors localize only the smaller L1 caches, and leave L2 as a shared resource. In this hierarchy, the overhead of *Jetty* would be comparable to just doing the regular L1 lookup, resulting in minimal benefit.

The *RegionScout* approach attempts to dynamically detect and filter out private regions of the memory space from performing the snoop-based coherence, reducing read induced tag lookups and write induced invalidation broadcasts [7]. As the authors write, this implementation is good for coarse-grained sharing, as reasonable power reductions only occur when the region size is rather large. The benefits of this approach are muted if the system has more fine-grained sharing or if the shared locations are not contiguous. Given the complexity and variability of applications that can be run on mobile processors, the assumption of fine-grained, contiguous data sharing cannot be guaranteed.

An application-driven solution was presented in [8], where the compiler or software developer would statically determine the shared regions of memory, allowing the remaining regions to be placed in a non-shared mode. Again, this solution utilizes rather coarse-grained regions and also relies on simple application algorithms for sharing. With more complex, desktop-like applications expected to run on modern mobile processors, the issues of dynamic memory and pointer access are introduced, rendering the compile-time approach insufficient.

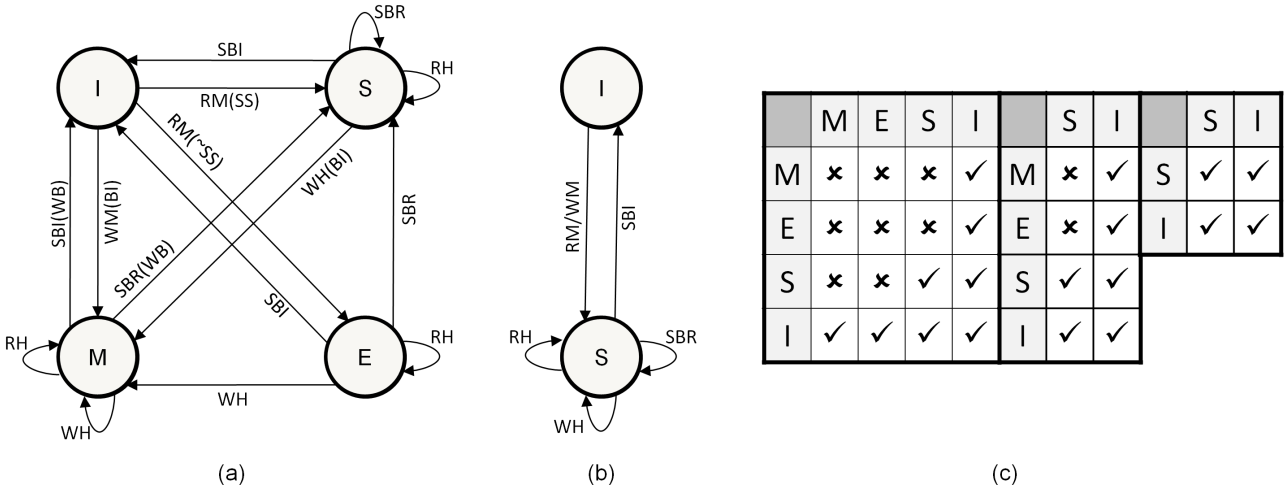
Moreover, since the cache is primarily present to mitigate the performance implications of long memory latencies, it is important to avoid causing significant degradation in performance just to recuperate power. Since having a longer run-time will ultimately lead to still more power usage of the overall system, there is an important balance that must be struck where the cache coherence power overhead is reduced, while not significantly degrading performance.

3. MOTIVATION

The typical cache coherency protocol commonly employed in MCSoC systems is MESI, consisting of four states per cache line [9]. The *Modified* (**M**) state implies that the data is the only copy within the localized caches and that that data is newer than the lower memory hierarchy. The *Exclusive* (**E**) state is similar to the **M** state, except that the data is clean (e.g. the same value as stored in the lower memory level). The *Shared* (**S**) state conveys that the data *may* also be in other local caches, and also that the data is clean. Lastly, the *Invalid* (**I**) state means no valid data is stored in that cache line.

Figure 3(a) shows the state transition diagram for the MESI protocol. Explanations of the various abbreviations used are listed in Table 1. Each cache line is annotated with a 2-bit representation of these four states. Data reads can be satisfied from any state except **I**. Upon a read miss, the cache line will enter either the **E** or **S** state, depending on the value present on the *Shared Signal (SS)*, which indicates whether at least one other cache also has the same data present. Data writes may only take place when in the **M** or **E** states, and only data in the **M** state requires a write-back operation when transitioned to another state or evicted. All valid states must snoop for any bus invalidate transactions, and upon matching the state will transition to **I**. Additionally, all valid states must snoop any bus read requests for matches in order to assert the *Shared Signal* on the bus, as well as to transition to the **S** state if currently in the **M** or **E** states.

Given this, if a read-miss occurs on the bus, the snoop controllers for all local caches must probe whether the requested data matches a valid entry and if that entry is in an



	M	E	S	I
M	✗	✗	✗	✓
E	✗	✗	✗	✓
S	✗	✗	✓	✓
I	✓	✓	✓	✓

	S	I
M	✗	✓
E	✗	✓
S	✓	✓
I	✓	✓

	S	I
S	✓	✓
I	✓	✓

Figure 3: (a) MESI Write-Back State Diagram; (b) Simplified Write-Through State Diagram; (c) Listing of Valid States Between Pair of Caches

exclusive state (**M** or **E**) to respond accordingly to maintain coherence. Alternatively, a write-miss or invalidation broadcast on the bus requires all other cache nodes to invalidate their corresponding entries for that given data address. This probing to identify whether an address sent on the shared bus is present in the local cache entails almost a full cache lookup; the tag arrays of the cache structure need to be accessed in order to fully resolve if the address is a match. Furthermore, the snooping logic occurs even if the data location is not being shared among other processor cores, leading to much wasted effort if the current application is primarily private-memory based and does very few shared reads or writes. These snooping requests are the major contributing factor to the excessive power consumption of hardware cache coherency protocols.

Furthermore, we found the local L1 caches exhibited unbalanced access patterns, wherein portions of the cache may be infrequently used at a given time, while other portions may be more heavily used and involved in multi-core data sharing. By responding dynamically to the application's run-time behavior, we can intelligently make modifications to the cache coherency protocol to appreciably reduce bus-snooping power overhead while minimally affecting execution performance. Indeed, by enabling a power-efficient implementation of hardware cache coherence, the execution performance will significantly improve compared to software-based or non-cacheable approaches. The goal of this paper is the dynamic identification of those cache lines which are highly accessed in a manner that would benefit from a full write-back MESI coherency protocol, versus those that can be in a simplified write-through configuration to help eliminate unnecessary read-based snooping overhead, thus conserving power while preserving most of the performance quality.

Abbreviation	Description
RM	Read Miss in local cache
RH	Read Hit in local cache
WM	Write Miss in local cache
WH	Write Hit in local cache
SBR	Snoop Bus Read from remote cache
SBI	Snoop Bus Invalidate from remote cache
(WB)	Write Back dirty data
(BI)	Send Broadcast Invalidate to all caches
(SS)	Shared Signal value currently on bus

Table 1: Listing of Abbreviations

4. IMPLEMENTATION

The proposed solution enables two complementary types of write behavior to occur per cache line: *write-back* and *write-through*. The benefits of a write-back policy become evident when there are numerous updates to a given local cache line. By avoiding writing each update back to the L2 cache, reductions in bus contention and L2 dynamic power are achieved. Unfortunately, cache coherence is essential for a write-back policy, as data in the shared L2 can become stale relative to newer data in a local cache. On the other hand, a fully write-through policy ensures coherence since copies in a given local cache will always match what is backed by the shared L2. However, a write-through policy suffers if there are many write operations, causing wasteful updates which congest the shared bus and waste power.

Our proposal is to allow a fine-grained, hybrid interaction between write-back and write-through functionality on a per-line basis, in a manner similar to [10]. When a given cache line is heavily modified in a short period of time, a write-back policy will deliver better performance. Conversely, if a given cache line is very infrequently updated, a write-through policy will deliver lower power by obviating much of the snoop-read overhead, while also not causing excessive bus contention due to write-misses. Furthermore, if that line frequently transitions from the **M** to the **S** state, a write-through policy is also desirable, as this behavior would be indicative of a producer-consumer algorithm where one core writes data that another core then reads in. In this case, the benefits from write-back with regard to multiple updates are muted, and a write-through setup will deliver equivalent performance while using less power.

To implement this solution, each cache line will be anno-

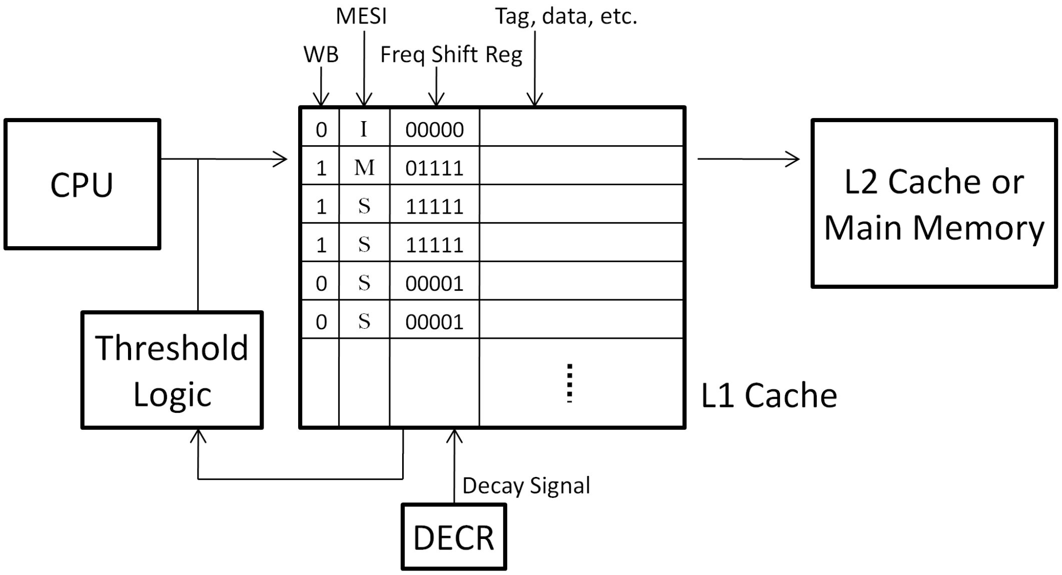


Figure 4: Proposed Cache Architecture

tated with a small number of additional bits to keep track of the new state information. Figure 4 provides a high-level view of the architectural additions. In particular, a small shift register is added to each cache line and will be used to measure the write frequency of that line. Upon accessing the cache line with a write operation, the *Frequency Shift Register* (*FSR*) is left shifted and fed a least-significant-bit (LSB) value of 1. Upon a write-back to L2 or 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 the line is frequently modified, saturate with all 0's if rarely modified or written back often, 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.

Furthermore, a *WB Enable Bit* will be added to each cache line, which will determine if the line is operating in write-back (1) or write-through (0) mode. When the line is placed in write-through mode, the MESI protocol is reduced to just two states, as shown in Figure 3(b): *Shared* and *Invalid*. Given that the MESI states correspond to bit values of 11, 10, 01, and 00, respectively, this write-through mode is achieved simply by doing an *AND* between the high-order bit of the MESI metadata and the *WB Enable Bit*. Figure 3(c) shows all the possible valid states of any pair of local caches, including both in write-back (MESI/MESI), mixed (MESI/SI), and both in write-through (SI/SI).

The architecture also has two global threshold registers: T_{WB_on} (*Write-Back On*) and T_{WB_off} (*Write-Back Off*). These threshold registers are the same size as the *FSR*, 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 write-back functionality based on the particular threshold value met.

To avoid ambiguity and deadlock, the threshold registers are constrained as follows: $T_{WB\_on} > T_{WB\_off}$. Thus, the minimum size of the threshold registers (and also the *FSR*) is 2-bits, and we will denote this size with N.

When the *FSR* reaches the T_{WB_on} threshold (due to a temporally heavy period of cache writes), the *WB Enable Bit* will be set to 1, enabling write-back functionality along with the full MESI coherency protocol. When the threshold falls below T_{WB_off}, the *WB Enable Bit* is set to 0, switching into a write-through mode. In this write-through mode, the coherence protocol will be degraded in order to glean power reductions. The cache line will still snoop for write-miss invalidations like before when in write-through **S** state, but the snooping for bus reads will be modified as described in the following section.

4.1 Modified Read-Miss Snooping Behavior

Normally the MESI protocol for a cache in the **S** state will snoop all read-misses and do a complete tag comparison to determine if the request matches in order to assert the *Shared Signal* on the bus. In contrast, during write-through mode the cache will always assert the *Shared Signal* for the given cache line regardless of whether the tags match. Thus, in write-through mode the costly tag comparisons during read-miss snooping are completely eliminated. In other words, during write-through mode the *Shared Signal* is simply equal to the cache line's state value (1 if **S**, 0 if **I**). Of course, this optimization can cause false positives if the requesting cache is in write-back mode. By a write-through cache always asserting the *Shared Signal* even if it potentially does not have the matching data, a correspond-

Thresholds	T_{WB_on}	T_{WB_off}
Config 0	00000010	00000001
Config 1	00001000	00000001
Config 2	00010000	00000100
Config 3	00100000	00001000
Config 4	10000000	00010000
Config 5	10000000	01000000

Table 2: Configuration Threshold Values

ing cache in write-back mode will always be forced to resolve a read miss by entering the **S** state instead of possibly the **E** state. It is important to observe that the **E** state is an optimization of the **S** state to reduce the bus invalidation overhead upon modification. Therefore, even if false positives occur, correctness will still be maintained. The only drawback is that if the write-back cache were to modify the data, since it was in the **S** state instead of the **E** state, it would need to issue a broadcast invalidate signal. An important observation is that this drawback only occurs on the first modification of a write-back line (the **S** to **M** transition). None of the subsequent writes will incur this penalty, since the line will now be in the exclusive **M** state.

An additional optimization is possible during a read miss in a write-through cache. The *Shared Signal* will be pre-set to 1 by the write-through cache during the bus read transaction, since it does not need to differentiate if any other caches also have a valid copy of the data (e.g. since there is no **E** state). During this bus read, any caches in write-back mode can leverage this if they are in the **S** state and bypass the tag lookup if the *Shared Signal* is already high. The reason this works is because if a MESI cache is in the **S** state, it will remain in the **S** state on a snooped bus read. Normally, the cache would also need to do the tag comparison to determine whether to assert the *Shared Signal* or not, but since the originating cache miss is in write-through mode, it will ignore the results of the *Shared Signal*. Thus, we can further reduce the overhead of read-miss snooping in the write-back caches as well in this situation.

4.2 Transitioning Between Policies

It is important to demonstrate that individual cache lines moving between the write-through and write-back policies will still maintain overall correctness and coherence. As described earlier, Figure 3(c) presents all the possible valid states of any pair of local caches. When transitioning from write-back to write-through mode, the behavior is the same as if a snoop bus read occurred. If the cache line is in the **M** state, a write-back is forced before transitioning into the **S** state. If the line is in the **E** state, the transition to **S** occurs without further ado. When transitioning from write-through to write-back mode, there is no special state changes that need to occur (e.g. the line will remain in the **S** or **I** state). As described earlier, the write-through mode will propagate any modifications to the L2 cache, maintaining coherence by still causing a write-miss on the shared bus to invalidate any copies that may exist in other local write-back or write-through caches. Since the write-through mode is essentially a simplified subset of the write-back MESI protocol (e.g. using only the **S** and **I** states), the coherence and correctness of the overall system remains intact even while corresponding cache lines in different cores move independently between the write-through and write-back modes.

Benchmark	Description
`ferret`	Content-based similarity search
`fft`	Discrete fast Fourier transform
`fluidanimate`	Incompressible fluid animation
`lu`	Dense matrix triangular factorization
`radiosity`	Equilibrium distribution of light
`radix`	Integer radix sort
`raytrace`	Three-dimensional scene rendering
`vips`	Image affine transformation
`x264`	H.264 video encoder

Table 3: Description of Benchmarks

The rationale for this approach is leveraging the uneven, temporal nature of cache line accesses. Some cache lines will feature many closely occurring write modifications, greatly benefiting from a write-back cache. Other cache lines may just be idle and unused in a given hot spot, and being in write-through mode can avoid wasting power conducting full read-miss snoops every time another core accesses the same index. Certain shared-memory multi-core applications will behave in a simple consumer-producer handshake fashion, wherein there are no performance benefits to using write-back on those cache lines, while other lines may incur numerous writes before being read by a remote core. The key principle is that we respond dynamically in a fine-grained, temporal fashion to exploit the strengths of the different cache write and coherency policies, while still maintaining overall performance.

5. EXPERIMENTAL RESULTS

In order to assess the benefit from this proposed architectural design, we utilized the M5 multiprocessor simulator [11], implementing a quad-core system similar to what is shown in Figure 2. We chose a representative advanced mobile smartphone system configuration, having per-core 32KB L1 data and instruction caches (1024-entry, direct-mapped with a 32-byte line size), and a shared 1MB L2 cache.

In addition to the baseline, classic write-back MESI cache architecture, we provide results for six different system configuration levels to dynamically transition lines between write-through and write-back mode, each increasingly more aggressive in favor of write-through mode. The associated threshold values for each configuration are available above in Table 2.

Nine representative shared-memory benchmark applications from the SPLASH-2 [12] and PARSEC [13] suites are used. A listing of these benchmarks and their respective descriptions is provided in Table 3.

In terms of storage overhead, the dynamically reconfigurable cache architecture would require the addition of $(N+1)$-bits to capture the *Frequency Shift Register* (*FSR*) and *WB Enable Bit* on each cache line, as well as N-bits for each of the two global threshold registers. Additionally, a decay countdown register (*DECR*) of 8-bits would be needed to generate a decay signal every 256 clock cycles. We chose an 8-bit *FSR* ($N = 8$), based on our observation that the 28 possible threshold pair combinations captured by 8-bits allows for a diverse span of ranges while not imposing prohibitive overhead. Given this and that we have 1024 lines in our implementation, this results in an additional 1155-bytes

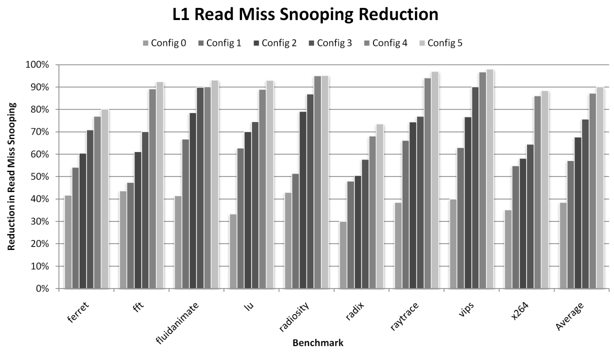


Figure 5: Reductions in L1 Read Miss Snooping

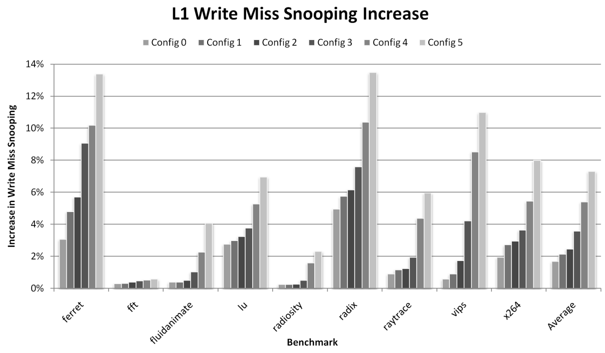


Figure 6: Increases in L1 Write Miss Snooping

(≈1.13KB) added to each L1 data cache; this overhead is a negligible amount of storage compared to the actual cache size of 32KB of data plus 2.38KB of tag and MESI metadata. Additionally, this implementation will only need to enhance the L1 data caches, as the instruction caches are typically read-only and would be implemented as write-through to avoid coherence. Thus, only minimal modifications of the cache subsystem are necessary to leverage this implementation.

Additionally, we chose an 8-bit *Frequency Shift Register* (*FSR*). This was based on our observation that the 28 possible threshold pair combinations captured by 8-bits allows for a diverse span of ranges, while not having prohibitive overhead.

Figure 5 shows the reductions to L1 read miss snooping by transitioning infrequently used cache lines into the simplified write-through mode. As cache lines are transitioned into this mode, they no longer need to process the costly tag lookups during a bus read event. Furthermore, any complement caches in write-back mode that have data in **S** state also avoid unnecessary tag lookups on read misses from the write-through cache. As one can see, *Config 0*, which is least aggressive in terms of entering write-through mode, still results in appreciable reductions to read miss snooping. As we move to more aggressive configurations, such as *Config 5*, one can see the majority of benchmarks begin to enjoy reductions in snoop tag lookup overhead near 90%. The two exceptions, *ferret* and *radix*, have very high rates of sharing and data exchange, and thus many of the cache lines do not ever fall below the T_{WB_off} threshold. The average reduction in read miss snooping overhead for the six configurations is summarized in Table 4.

While the reductions to read miss snooping will be the major contributing factor to reducing the overall cache power utilization, we must also take into account the increase in write miss snooping that will inherently occur as more cache lines are placed in write-through operation. Write-through mode fundamentally will cause more writes to be placed on the shared bus, resulting in more frequent broadcast invalidations (write misses). All local cache lines in **S** mode will need to then snoop to ensure they invalidate their copy of the data if present. Fortunately, an important attribute of this proposed architecture is to only allow the *FSR* to decrement when write operations are infrequent, or when there are many write-backs. In this fashion, we enter the write-through mode only when the quantity of write operations is statistically improbable, lessening the chance of attenuating our gains in power reduction with additional overhead in write miss snooping. Additionally, applications are typically dominated by more read operations than writes, which will further help tip the scales in our favor.

To demonstrate these observations, Figure 6 shows the increases to L1 write miss snooping. As one can see, by moving from a less aggressive write-through threshold (*Config 0*) to a more aggressive one (*Config 5*), the amount of write-miss-induced snooping grows rapidly. The average increase in write miss snooping for the six configurations is shown in Table 4. One can begin to see how forcing write-through mode aggressively can start to diminish the returns in power savings from avoiding read miss snooping. Furthermore, the additional write misses incur the power overhead of having to write the data back into the L2 cache, as well as introducing increased pressure on the shared bus. Both of these factors need to be taken into account when considering the overall cache subsystem power usage.

Configuration	Read Reduction	Write Increase
Config 0	38.50%	1.67%
Config 1	57.16%	2.13%
Config 2	67.69%	2.45%
Config 3	75.71%	3.57%
Config 4	87.26%	5.39%
Config 5	90.06%	7.30%

Table 4: Average Read Miss Snooping Reductions and Write Miss Snooping Increases

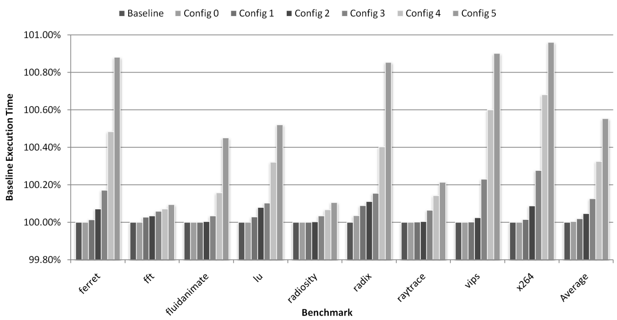


Figure 7: Global Execution Time Overhead

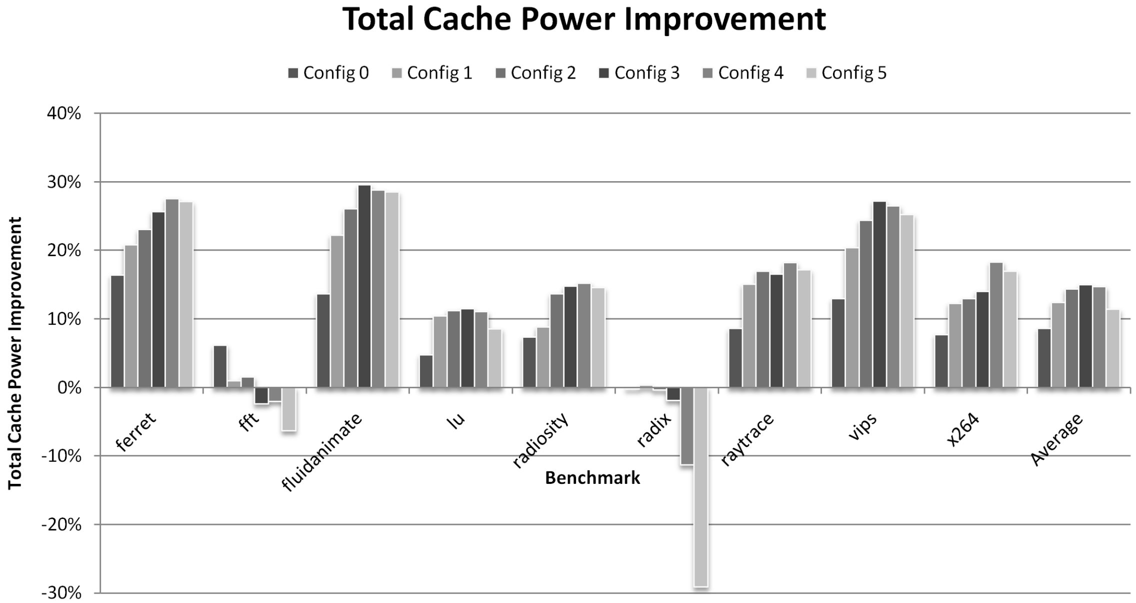


Figure 8: Total Cache Subsystem Power Improvement

Benchmark	*Config 0*	*Config 1*	*Config 2*	*Config 3*	*Config 4*	*Config 5*
`ferret`	16.41%	20.84%	23.05%	25.67%	27.59%	27.11%
`fft`	6.16%	0.98%	1.52%	-2.39%	-2.02%	-6.31%
`fluidanimate`	13.69%	22.22%	26.11%	29.58%	28.82%	28.51%
`lu`	4.78%	10.45%	11.25%	11.52%	11.10%	8.55%
`radiosity`	7.36%	8.85%	13.71%	14.78%	15.25%	14.56%
`radix`	-0.19%	0.36%	-0.36%	-1.89%	-11.30%	-29.12%
`raytrace`	8.60%	15.07%	16.99%	16.56%	18.24%	17.15%
`vips`	12.95%	20.39%	24.44%	27.19%	26.50%	25.23%
`x264`	7.72%	12.29%	12.97%	14.05%	18.31%	16.95%
`Average`	8.61%	12.38%	14.41%	15.01%	14.72%	11.41%

Table 5: Total Cache Power Improvement

Figure 7 provides the execution time across the benchmark's lifetime. The impact of additional bus contention due to increased write miss transactions is taken into account, and the impact to the overall run-time can be observed. As one can see, on average there is negligible run-time overhead when employing *Config 0*, *Config 1*, or *Config 2*, but with the more aggressive *Config 4* and *Config 5* there is a small overhead of approximately 0.32% and 0.55%, respectively. The highest overhead was observed in the *x264* benchmark, having an increase of 0.96%. Luckily, the write-through operation itself does not introduce a timing cost for the processor, as it occurs seamlessly in the background.

With regard to overall cache power efficiency, our primary ambition for these cutting-edge 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 1155-bytes (≈1.13KB) of additional storage elements, along with some necessary routing signals and muxing.

We used CACTI [14] and eCACTI [15] to estimate the overall power consumption for the entire cache hierarchy, including the proposed modifications. Furthermore, we take into account the additional power incurred while doing extra L2 cache writes (due to write-through operation), as well as the power associated with the prolonged run-time due to bus contention. 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.

Figure 8 shows the total cache subsystem power improvement across all six configurations for the complete execution of each benchmark. A tabular listing is provided in Table 5. As one can see, there is a continuum across the different configurations. As one progresses from *Config 0* to *Config 3*, on average the benefits of power optimization increase. This is due to the increase in read miss snoop minimization. However, there is a point of diminishing returns as the biasing in favor of write-through mode increases in aggressiveness. Continuing on to *Config 4* and *Config 5*, one can see the power benefits begin to wane. This is due to the increase in L2 and bus activity due to more frequent write-backs caused by the aggressive policy. Also, it can be observed that some applications are particularly susceptible to poor power performance with the more aggressive configurations. In particular, we found *fft* and *radix* to be particularly susceptible, mainly due to the fact that these algorithms have close to the same number of writes as reads.

Nevertheless, all six configurations still provide an average overall improvement to power utilization, and *Config 3* appears to do the best across all benchmarks. The key is to find the ideal balance between the overhead of write miss snooping and the reductions in read miss snooping.

6. 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.

7. CONCLUSIONS

As shown, caches contribute a significant amount of the overall processor power budget. 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. Moreover, these smartphones are beginning to enter the multi-core SoC domain, and cache coherence will further exacerbate the power overhead of caches. 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, multi-core mobile processors. By using varying configurations, a notable increase in memory performance can be achieved by enabling complete hardware MESI cache coherency when plugged-in or fully-charged, while a significant decrease in power usage can be achieved when the user switches the device into a low-power mode, wherein much of the read miss snooping overhead can be obviated with little impact to performance. This has been demonstrated by using an extensive and representative set of shared-memory simulation benchmarks, and six example configurations. The proposed technique has significant implications for mobile processors, especially high-performance, power-sensitive devices such as smartphones. It significantly reduces power consumption while minimally degrading run-time performance, which in turn will permit full hardware cache coherence to be integrated into future mobile processors without the associated power overhead cost.

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.

8. REFERENCES

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

[2] N. 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] G. Bournoutian and A. Orailoglu. Miss reduction in embedded processors through dynamic, power-friendly cache design. In *DAC '08: Proc. 45th Conf. on Design Automation*, pages 304–309, New York, NY, 2008.

[4] M. Ekman, P. Stenström, and F. Dahlgren. TLB and snoop energy-reduction using virtual caches in low-power chip-multiprocessors. In *ISLPED '02: Proc. 2002 Intl. Symposium on Low power Electronics and Design*, pages 243–246, New York, NY, 2002.

[5] C. Saldanha and M. Lipasti. Power efficient cache coherence. In *Proc. Workshop on Memory Performance Issues*, 2001.

[6] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. JETTY: Filtering snoops for reduced energy consumption in SMP servers. In *HPCA '01: Proc. 7th Intl. Symposium on High-Performance Computer Architecture*, pages 85–96, Los Alamitos, CA, 2001.

[7] A. Moshovos. RegionScout: Exploiting coarse grain sharing in snoop-based coherence. In *ISCA '05: Proc. 32nd Intl. Symposium on Computer Architecture*, pages 234–245, Washington, DC, 2005.

[8] A. Dash and P. Petrov. Energy-efficient cache coherence for embedded multi-processor systems through application-driven snoop filtering. In *DSD '06: Proc. 9th EUROMICRO Conference on Digital System Design*, pages 79–82, Washington, DC, 2006.

[9] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA '84: Proc. 11th Intl. Symposium on Computer Architecture*, pages 348–354, New York, NY, 1984.

[10] G. Bournoutian and A. Orailoglu. Dynamic, non-linear cache architecture for power-sensitive mobile processors. In *CODES/ISSS '10: Proc. 8th Conf. on Hardware/software codesign and system synthesis*, pages 187–194, New York, NY, 2010.

[11] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: modeling networked systems. *IEEE Micro*, pages 52–60, 2006.

[12] J. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *SIGARCH Computer Arch. News*, pages 5–44, 1992.

[13] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *PACT '08: Proc. 17th Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 72–81, New York, NY, 2008.

[14] S. J. E. Wilton and N. P. Jouppi. CACTI: An enhanced cache access and cycle time model. *IEEE Journal on Solid-State Circuits*, pages 677–688, 1996.

[15] M. Mamidipaka and N. 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.