Exploiting Access Semantics and Program Behavior to Reduce Snoop Power in Chip Multiprocessors

Chinnakrishnan S. Ballapuram[†] Ahmad Sharif Hsien-Hsin S. Lee

School of Electrical and Computer Engineering
Georgia Institute of Technology, Atlanta, GA 30332

FDC Architecture, Intel Corporation[†]
chinnak@ece.gatech.edu, ahmad@gatech.edu, leehs@gatech.edu

Abstract

Integrating more processor cores on-die has become the unanimous trend in the microprocessor industry. Most of the current research thrusts using chip multiprocessors (CMPs) as the baseline to analyze problems in various domains. One of the main design issues facing CMP systems is the growing number of snoops required to maintain cache coherency and to support self/cross-modifying code that leads to power and performance limitations. In this paper, we analyze the internal and external snoop behavior in a CMP system and relax the snoopy cache coherence protocol based on the program semantics and properties of the shared variables for saving power. Based on the observations and analyses, we propose two novel techniques: Selective Snoop Probe (SSP) and Essential Snoop Probe (ESP) to reduce power without compromising performance. Our simulation results show that using the SSP technique, 5% to 65% data cache energy savings per core for different processor configurations can be achieved with 1% to 2% performance improvement. We also show that 5% to 82% of data cache energy per core is spent on the non-essential snoop probes that can be saved using the ESP technique.

Categories and Subject Descriptors B.3.2 [*Design Styles*]: [Cache memories]

General Terms Design, Experiment, Power, Performance

Keywords Chip Multiprocessors, Internal and External Snoops, Self-Modifying Code, and MESI protocol

1. Introduction

The continuous miniaturization of devices has brought chip multiprocessors (CMPs) into all market segments ranging from servers to mobile products. In addition to improving performance, CMPs can also be used to address emerging issues such as security [32], reliability [18, 34], etc. In a CMP system, cache coherence maintenance is a complex task, and the support for self-modifying code (SMC) and cross-modifying code (XMC) further increases its design complexity. There are two kinds of snoops in a CMP-SMP system (CMP-based SMP system), internal and external snoops. The *internal snoops* are triggered and responded to by cores within the *same* CMP, while the *external snoops* are triggered and responded to by *different* CMPs in a CMP-SMP system. Prior works have dealt with the analysis and optimization of external snoops in an SMP environment, but have limited analysis in internal snoop behavior, which includes the self-modifying and cross-modifying code snoops in a CMP. The snoop response time is becoming critical in a CMP system for the following reasons: 1) increase in number of cores per die, 2) continuous extension of the vertical cache hierarchy, and 3) increase in size of the cache and load/store buffers. It is further exacerbated by the conservative nature of the cache coherence protocol to send snoop probes for all variables in the program without distinction. Additionally, the requirements to send snoop probes differ based on the cache inclusion policies, leading to different response times.

Cache inclusion properties [7] provide design guidelines for cache hierarchies and to facilitate cache coherence implementation. Strongly inclusive caches are generally used in academic research, while commercial processors implement both strongly inclusive and weakly inclusive policies. For example, IBM's Power5 uses strongly inclusive caches [7], Intel Pentium Pro uses weakly inclusive caches [23], and both Compaq Piranha [9] and AMD Athlon use exclusive caches in their designs. All three cache policies have pros and cons. In exclusive caches, data resides in only one of the caches in the hierarchy to increase the effective cache capacity. In both weakly inclusive and exclusive policies, all snoop requests need to be propagated from the last level cache to all the cores' lower level caches (i.e., caches nearer to the core) and other related memory buffers in a CMP. This snooping-all technique may not scale well with an increasing number of cores as it increases power and inter-core communication. In strongly inclusive cases, snoops probe all the lower level caches only when a cache line is present in the last level cache to obtain the most recent version from the owner core. An alternative to avoid sending snoop probes to all cores in inclusive caches is to add a *shadow* set of all lower level cache tags alongside the shared last level cache to maintain coherency. Unfortunately, the overheads to maintain these shadow tags become higher as the number of cores, cache hierarchy, and size of the lower level caches increases.

In general, cache coherence protocols are implemented in a conservative manner. They always assume that all variables used in a program are shared with other concurrent threads or programs. To maintain functional correctness, all reads and writes that reach the last level cache must send snoop probes to the other cores or processors in the system. However, some cores do not need to be snooped if we know in advance that certain variables based on their program semantics will get a miss response. The user input, programming languages used to design an application, and differentiation

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.

ASPLOS'08 March 1–5, 2008, Seattle, Washington, USA.
Copyright © 2008 ACM 978-1-59593-958-6/08/03. . . $5.00

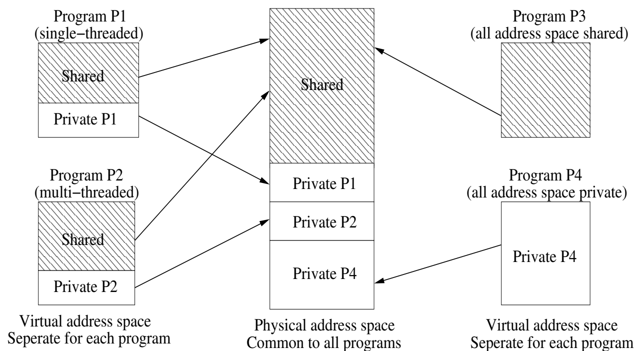

Figure 1. Various program categories.

between single and multi-threaded applications can play an important role in determining the number of snoop probes generated. The efficiency of the snoop probes can be highly improved by utilizing the programming language constructs and improving the contract between the user and the application. The two main reasons for cache coherent protocol and its implementation to be conservative are that: 1) cache coherence protocol does not distinguish between the shared variables and the non-shared ones, and 2) when a thread migrates from one core to another because of OS scheduling, it typically leaves behind its old work (e.g. modified variables) in the old caches and is required to snoop all the cores to continue its work in the new core. We try to address these drawbacks with support from both the hardware and software. Our goal is to relax the conservative nature of cache coherency protocol and its implementation by selectively sending snoop probes for only certain program variables to reduce the excessive bandwidth and other resources they use.

The rest of this paper is organized as follows. Section 2 discusses program semantics. Section 3 describes different snoop probes in a typical CMP system. Section 4 proposes Selective Snoop Probe (SSP) and Section 5 proposes a method to handle snoops in the event of thread migration. Section 6 details the Essential Snoop Probe (ESP). We analyze our experimental results in Section 8. Related work is discussed in Section 9. Finally, Section 10 concludes.

2. Program Variables and Snoop Probes

Figure 1 illustrates four different program categories based on the nature of data shared in the program. Programs P1 and P2 that contain both the shared and private variables are similar. The difference between them is that while P1 is a single-threaded program that shares its global variables with other *programs* in the system, P2 is a multi-threaded program that shares its variables with other *threads and programs*. The third category is represented by program P3, where all variables are assumed to be shared with other programs, which is normally the case assumed by a cache coherence protocol implementation. The fourth category is represented by P4, where all the variables are local to the process without any sharing.

Figure 2 shows the organization of variables in a typical single and multi-threaded application corresponding to programs P1 and P2. In general, both code and data are assumed to be shared with other programs in the system. This region is marked as *shared* in Figure 1. On the other hand, registers and stack, which are not globally visible to other programs or the outside world, are local to each thread. The variables in this region are shown as *private* for P1, P2, and P4 in Figure 1.

The knowledge about the semantics of variables in a program provides an opportunity to optimize snoops in a shared memory

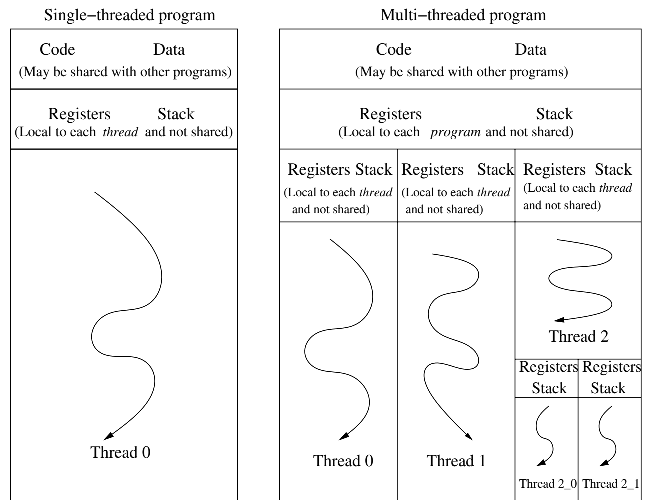

Figure 2. Thread stack in single- and multi-threaded programs.

environment. Current cache coherence implementations do not differentiate between single and multi-threaded programs that can coexist in a CMP. Sometimes a single-threaded program may not use any shared memory constructs. In this case, the program might be "self-sufficient" in a cache coherent sense, where reads and writes need not probe other cores or processors in the system. This information can potentially be identified during program compilation. Also, programmers can give their input or can be identified during compilation that the program does not contain self-modifying code. These inputs are valuable to the underlying processor for minimizing those ineffectual snoops generated by the snoop controller, thereby achieving an overall improvement in system power and performance.

We first propose a hardware-only technique called Selective Snoop Probe that exploits the properties of stack variables to filter out unnecessary snoops. Then we propose a hybrid hardware/software technique called Essential Snoop Probe that provides the necessary architectural support to reduce the number of snoop probes based on the programming language and compiler hints for all types of program variables. Both techniques can be adopted by all types of inclusive/exclusive cache policies.

3. Snoop Classification

The internal snoops to lower level caches are inevitable in a CMP system, where several cores are on the same die. They are not only necessary to maintain cache coherency but are also required to support self-modified code (SMC) and cross-modified code (XMC). In this section, snoops resulting from SMC/XMC, snoop flows, snoop probes, and triggers in a typical CMP are described.

3.1 Snoops due to self/cross-modifying code

Self-modifying code (SMC) changes its own instructions by rewriting to them. Many commercial processors including IA-32, Itanium and IBM POWER support self/cross-modifying code. There are many applications that use this feature; one example is the Just-In-Time compilers that use SMC to generate optimized codes at runtime. To provide this support, on every write to a memory location in a code segment that is currently cached or prefetched, the processor should invalidate the associated cache line in the instruction cache and prefetch buffers. For example, in the Pentium 4 processor, when a write to a code segment matches the target instruction that is already decoded and resident in the trace cache, the entire trace cache is invalidated. To detect such behavior, each

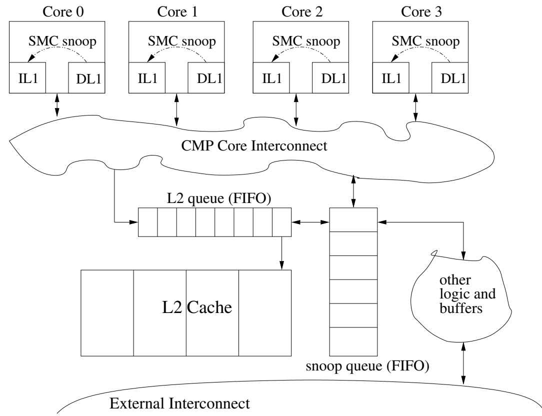

Figure 3. Snoop flows in a Quad-core CMP system.

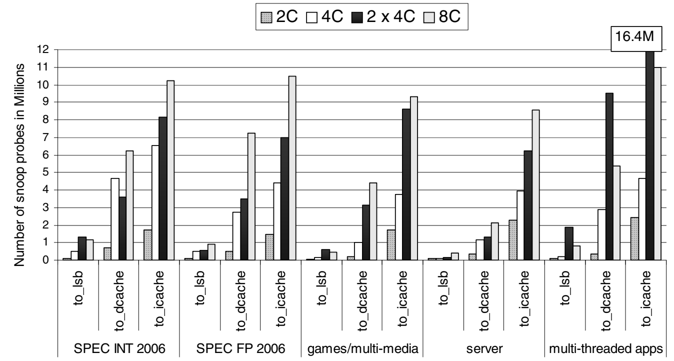

Figure 4. Snoop probes in different benchmark suites.

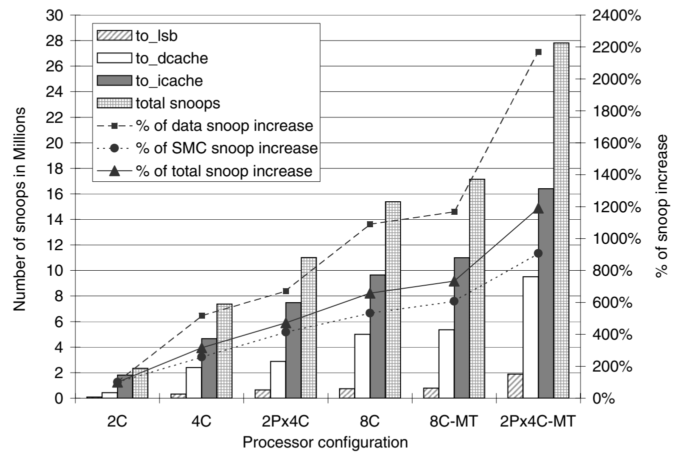

Figure 5. Snoop probes and snoop rate for different processor configurations.

store address needs to send a snoop probe to the instruction cache. When the snoop probe address matches a cache line in the instruction cache or prefetch buffer, the instruction cache invalidates the corresponding cache line. In addition, upon every instruction access to the last level cache, snoop probes to the same core's data cache and store buffers are sent to support the correct behavior of the SMC. Similarly, snoop probes to the other cores in CMPs are sent to support cross-modified code (XMC).

3.2 Snoop flows

Figure 3 shows the flow of internal and external snoops in a typical quad-core CMP. Each core has private L1 instruction (IL1) and data caches (DL1), and all four cores share one unified last level L2 cache. The L2 is accessed on any IL1 or DL1 miss, L1 and L2 prefetchers, and external snoops in an MP system. The CMP interconnect that connects all lower level cores can be a bus, ring, or arbiter-based interconnect. The requests that need to access the pipelined L2 cache are queued in a common hardware structure called the *L2 queue*. The internal and external snoop requests are queued in another hardware structure called the *snoop queue* [24, 13]. A snoop queue entry is allocated during an access to the last level cache or whenever an external snoop request is received. Each entry in the snoop queue spawns snoop probes to all the cores' IL1 and DL1, load/store buffers, and MSHRs, based on the type of each memory request. Note that each snoop request allocated in the snoop queue spawns multiple snoop probes to different hardware structures of all cores. The snoop response, and data if necessary, are propagated to the requesting core. It is also necessary to perform a snoop queue match before sending responses to the external bus and before writing a cache line to the last level cache for requests reaching it to maintain cache coherency and memory consistency. Thus, it becomes crucial to reduce the occupancy of the snoop queue for performance considerations.

Snoop flows differ based on the type of CMP interconnect architecture used. In a ring interconnect, snoops are sent across the ring and all requests to the cores are queued and processed. The cores return a snoop response, and data if applicable, over the ring. The snoops can consume a large amount of bandwidth if not properly handled or optimized. Regardless of the type of implementation, it is important to complete snoop probes and return responses to the requesting core as quickly as possible or to avoid snoop probes completely whenever possible to improve the overall system performance.

3.3 Snoop triggers and snoop probes

Table 1 based on [4] shows internal snoop trigger points and hardware structures that need to be snooped to support cache coherency and S/XMC snoops. The first column shows incoming requests to the last level cache (LLC), in this case the L2. The next two columns correspond to hardware structures of the core that triggers the event and the rest of the columns correspond to the rest of the CMP cores that respond to these snoop probes. The entries in the last row show that on every store address, the instruction cache of the corresponding core in the front-end needs to be snooped to support self-modified code. The reads and Requests For Ownership (RFO) need to probe at least three to four (this equals the number of relevant hardware structures) times the number of cores in the CMP to maintain cache coherency. A code fetch reaching the LLC can be much more expensive than a data read or RFO as it sends snoops to all modules, shown in the snoop table, but is mitigated by lower instruction cache miss rates. This table clearly shows that as the number of cores per die and vertical cache hierarchy increases, these internal snoops will likely become the bottleneck in performance and power consumption.

Figure 4 and Figure 5 show the number of snoop probes received by each hardware structure for various benchmark categories with different processor configurations. The graph shows the results collected from around 200 traces for 6M instructions each based on the simulation methodology described in Section 8. Each core in all configurations has split IL1 and DL1. All cores in one

Incoming events to LLC	IL1 of this core	DL1 of this core	LSB of this core	DL1 MSHR, WBB of this core	IL1 of the other 3 cores	DL1 of the other 3 cores	LSB of the other 3 cores	DL1 MSHR, WBB of the other 3 cores	shared L2 queue
RFO	-	Event Trigger	-	-	XMC snoop to invalidate line	snoop	snoop load buffer only to invalidate	snoop to invalidate pending requests	snoop to invalidate
Data Read	-	Event Trigger	-	-	-	snoop	-	snoop	snoop
Code Fetch	Event Trigger	SMC snoop	snoop store buffer only (updated writes)	snoop (updated writes)	-	XMC snoop	snoop store buffer only (updated writes)	snoop	SMC snoop
Shared L2 evict	-	snoop	-	snoop	-	snoop	-	snoop	snoop
On every store address dispatch	SMC snoop to iL1	-	-	-	-	-	-	-	-

Table 1. Snoop triggers and snooped units in a quad-core system.

processor share the same L2. Also note that L2 employs a weakly-inclusive policy, where a cache line in L1 may not be present in L2. Figure 4 shows the number of snoops to three microarchitecture modules, namely, Load-Store Buffer (LSB), DL1, and IL1. The figure illustrates that the number of snoops to the IL1 (to_icache) to support self-modifying code is dominant, followed by those to the data cache and those to the LSB.

Figure 5 shows the aggregate number of snoops for 6 different configurations for the same benchmark in Figure 4. The 2C, 4C, and 8C configurations represent two, four, and eight-core CMP systems, respectively, on which two, four, and eight copies of a single-threaded program are run. The 2Px4C configuration is a system with two processors, where each processor has four cores. Each core runs one copy of a single-thread application. The 2Px4C-MT contains two quad-core processors running an 8-way multi-threaded application while 8C-MT is simply an 8-core system running the same multi-threaded application. Figure 5 shows that the number of snoops steadily increases with the number of cores. The multi-threaded (8C-MT) workload incurs a slightly higher number of snoops than its single-threaded counterpart as the shared variables between threads increase snoop probes. Also, there is a slight increase in the snoop traffic as a result of the external snoops in a dual-processor configuration 2Px4C compared to the uni-processor configuration 4C. Also, there is a high increase in the number of snoops when a multi-threaded workload is run on a dual-processor (2Px4C-MT) configuration because of shared variables and external snoops. The secondary axis of Figure 5 shows the percentage snoop increase with respect to the 2C configuration as the baseline. Although both Figure 4 and Figure 5 show that the number of snoops to the IL1 is high, the rate of data snoop increase is much higher compared to the instruction cache, as shown by the secondary axis of Figure 5. As the number of cores increases beyond 8 or 16 cores, snoop probes to the data cache and LSB will limit the performance. The number of snoops tends to increase in multi-threaded applications and is further aggravated in an MP, limiting the performance improvement gained by parallelizing the applications. We also observed that many of these snoop probes get clean responses from the cores. The main reason is that the knowledge about shared variables and the nature of the application is not conveyed to the processor. To address these shortcomings, we proposed a hardware technique called Selective Snoop Probe (SSP) and a compiler-based hardware supported technique called Essential Snoop Probe (ESP) that use the properties of variables used in the program. These two techniques relax the conservative nature of the cache coherency protocol and snoop selectively to achieve better power and performance.

4. Selective Snoop Probes (SSP)

In this section, we describe and discuss our hardware solution, which selectively filters snoop probes for requests reaching the last level cache (LLC). Each access to the LLC spawns snoop probes, as described in Table 1, to keep the data coherent in all the cores. The requests that reach the last level cache can be divided into two types based on the region of memory accessed: stack accesses and non-stack accesses. The reason we partition accesses into these two types is based on a simple observation that snoop probes generated as a result of stack access requests should always receive a clean response from other cores as stack memory is considered private to its own thread. The snoop probes generated by non-stack accesses can be further divided into two types: those receiving a hit or modified response, and those receiving a clean response. The snoop probes caused by requests that access the stack and those that receive a clean response by non-stack accesses are candidates that can be eliminated. We observed that for all benchmarks that the number of positive (hit or modified) responses from the cores that respond is, in fact, is much fewer than the number of snoop probes sent. The two main reasons for receiving clean responses are: 1) the snoop probes to local thread stack variables are clean, and 2) programs typically do not contain self-modifying code. Unfortunately, modern day architectures do not differentiate these types of variables, resulting in a large number of unnecessary snoops. These snoops and their clean responses consume power to communicate the transactions and to probe the hardware structures. It can also affect performance when a dedicated snoop port is not implemented in the core for area reasons. To address these wasted snoop probes, we propose a simple hardware technique called *Selective Snoop Probe* (SSP). It uses a stack-bit (S-bit) annotation to eliminate snoop probes caused by stack accesses. Meanwhile, MESI-state-based counting Bloom filters are proposed to eliminate snoop probes caused by non-stack accesses that give clean responses.

Another interesting behavior exhibited by programs is that stack accesses typically account for a considerable portion of all memory references [22, 20, 21]. We profiled several benchmark suites to determine the distribution of memory accesses that reach the LLC. As Figure 6 shows, about 30% of memory references go to stack using the stack or frame pointer as the source or destination register which is the dominant method to access the stack in IA-32 codes. By decoding the source or destination register identifier, a processor can isolate memory instructions that go to the stack. To enable this isolation in hardware, we propose to annotate load and store instructions with a single bit in the decode stage to indicate whether an access is going to stack. Each request reaching the LLC carries this annotation as part of the operation.

Figure 7 shows the details of internal and external snoop filtering mechanisms based on the program semantics of the variables.

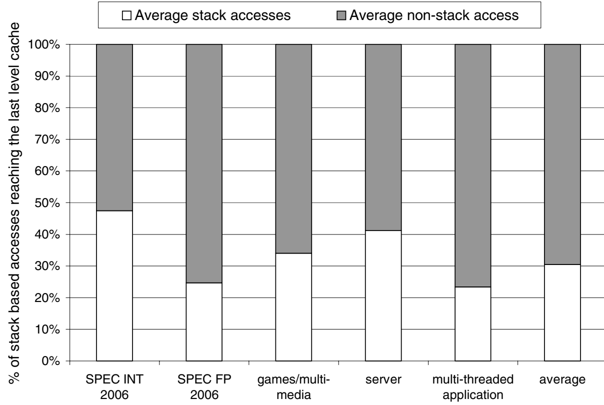

Figure 6. Stack accesses to the last level cache.

The SSP technique uses stack-bit (S-bit) annotation for stack accesses and counting Bloom filters for non-stack accesses to selectively filter out snoop probes that are not needed. Recently, Bloom filters [10] have been widely used in various microarchitecture optimizations for performance and power [26, 31, 17, 29]. A Bloom filter is a probabilistic data structure used to indicate if an element is a member of a set. Since it guarantees no false-negatives, it is used as an efficient structure to represent a large data set in a compressed signature form. In our SSP implementation, two distinct counting Bloom filters labeled ① and ② in Figure 7 are added to each core. The first one tracks the valid cache lines in the instruction cache to eliminate unnecessary SMC snoops. The second one tracks the data cache lines to eliminate the snoops that receive clean responses from non-stack accesses. The updates to counting Bloom filters are denoted by lines marked u1 and u2, and reads are denoted by lines marked r1 and r2 in Figure 7. The operation of SSP is divided into three main parts: SSP for SMC, SSP for stack accesses, and SSP for non-stack accesses.

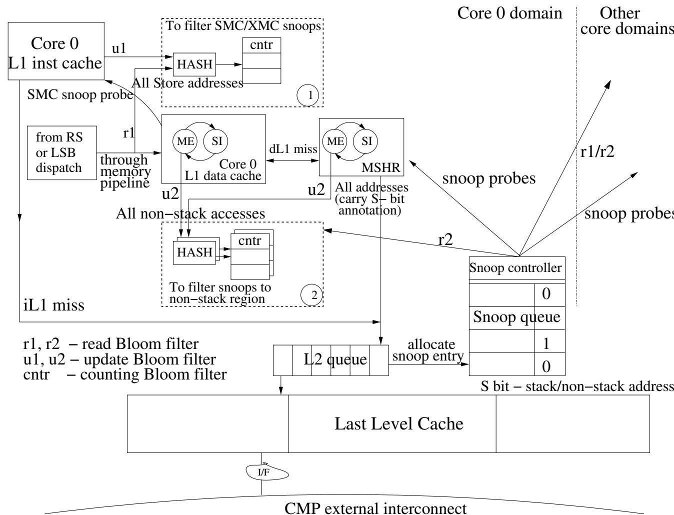

Figure 7. Selective Snoop Probes (SSP).

4.1 Selective snoop probe for SMC (SSP-SMC)

The counting Bloom filter inside the box labeled ① in Figure 7 tracks all valid lines in the I-cache. Whenever an I-cache line becomes invalid, it is removed from this Bloom filter. The insertion

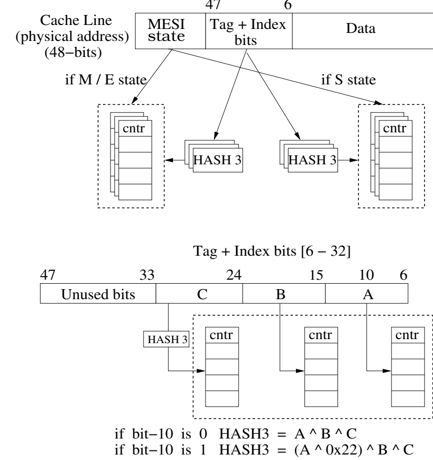

Figure 8. Hash functions used in counting Bloom filters.

and removal of addresses in the Bloom filter is denoted by line u1. The data cache control unit first looks up this Bloom filter (denoted by line r1) on a store address dispatch by the Reservation Station (RS) or Load-Store Buffer (LSB). The SMC snoop probe to the instruction cache is sent only when a lookup to the Bloom filter generates a hit. Thus, unnecessary SMC snoop probes to the I-cache are eliminated.

4.2 Selective snoop probe for stack region

The requests that reach the LLC carry stack-bit (S-bit) annotation as part of the operation. As described earlier, the S-bit annotation is set in the decode stage based on the source and destination register identifiers. The snoop controller checks the S-bit before spawning snoop probes to determine if it is necessary to do so. If the S-bit is set, the snoop controller do not send any snoop probes. The stack access requests do not use any Bloom filter and use only the S-bit annotation for their operation. The stack access requests constitute 30% of the LLC accesses (Figure 6), for which snoop probes are eliminated by checking the S-bit.

4.3 Selective snoop probe for non-stack region

On average 70% of requests reaching the LLC go to the non-stack region. The counting Bloom filter inside the box labeled ② in Figure 7 tracks all non-stack accesses that look up data cache and MSHR, denoted by lines marked u2. The snoop controller looks up the counting Bloom filters in all the other cores based on Table 1 and gathers information before spawning snoop probes (denoted by lines r2 and r1/r2). This information is obtained only for non-stack accesses. The snoop probes to the instruction and data caches are spawned only for those accesses that are needed based on the information gathered earlier. The snoop controller may still send some snoop probes that are unnecessary as Bloom filters can miss those lines due to aliases, resulting in false-positives. As the Bloom filter only maintains information for non-stack addresses, this inaccuracy caused by aliasing has been largely reduced. The design of counting Bloom filters, the effectiveness of hash functions, and the advantages of using the SSP technique are discussed in the following sections.

4.4 Bloom filter and hash function

The counting Bloom filter inside box ② in Figure 7 records non-stack accesses based on the state transition of the MESI protocol. The invalidation-based MESI cache coherence protocol is used in this work. The Bloom filter for the data cache is divided into two

sets. One tracks the M(odified) / E(xclusive) states together, and the other tracks S(hared) state. The ME-Bloom filter records the signature when the MESI state of a cache line is transitioned to the M/E state. Similarly, when the cache line state is transitioned to S, it is recorded in the S-Bloom filter, as shown in Figure 8. The reason for segregating Bloom filters is to reduce aliases and decrease the number of false-positives. This is based on the observation from all benchmarks that more snoops take cache lines to S states than to M or E states, as load instructions are executed more frequently than stores.

The hash functions for counting Bloom filters in boxes ① and ② use cache line address bits to index the Bloom filter arrays. Three counting Bloom filter arrays of 512 entries each are used as illustrated in Figure 8. Note that the hash functions are fixed. We did not use different hashes for different benchmark programs. The first array is indexed directly by lower-order bits [14:6]. The second array is indexed by bits [23:15] of the physical address. The third array is indexed by XOR-ing bits [14:6], bits [23:15], and bits [32:24] if bit 10 is 0. Otherwise, bits [14:6] are XOR-ed with 0x22 instead of directly using the bits [14:6], as shown in Figure 8. This combination of hashing is done to distribute the indexing of addresses to all entries for alias reduction. The Bloom filter has 10-bit counters, as there are 64 sets and eight ways in the 32KB cache. The 10-bit counters will ensure that even if the hash function mapped all lines in the cache to the same Bloom filter entry, there would still be no overflow. In reality, our hash function is good enough to uniformly spread the cache lines over large sets in most cases. The Bloom filter counters are cleared and reset when modified lines are written back to the last level cache, as all lines in the L1 will become invalid after eager writeback.

The number of snoop probes eliminated by the SSP technique is sensitive to the size of the Bloom filter array. The false-positive rate decreases as array size increases. The average false-positive rate we accrued using 512 entries varied from 14% to 35% for different applications. Note that these statistics do not faithfully represent the exact false-positives in reality; in fact, they are overly pessimistic. The false-positives progressively increased as the program continued execution. Nonetheless, in real scenarios, the Bloom filter will be cleared upon context switches, alleviating the false-positives.

In summary, there are several advantages of using the SSP technique. First, stack accesses from each local core do not snoop other cores. Second, non-stack access induced snoops are selectively propagated when identified as necessary by counting Bloom filters. Third, the front-end of the core that includes the instruction cache and prefetch buffers is snooped only when necessary. The SSP technique thus eliminated many of the unnecessary snoop probes for non-stack addresses and all snoop probes for stack accesses, reducing power and improving performance. The external snoop requests also go through snoop queue allocation and the same procedure described above is followed.

5. Eager Writeback to Last Level Cache

The two main reasons to send snoop probes for requests reaching the last level cache to all cores are: 1) the cache coherence protocol is conservative as it assumes all variables in the program are shared, and the underlying processor follows this conservative implementation to broadcast snoop probes, and 2) when a thread migrates from one core to another because of OS scheduling, it typically leaves behind its modified variables and requires to snoop all cores/processors later to obtain correct data in a physically indexed and tagged cache implementation. The SSP technique described previously addresses the first condition by relaxing the conservative snoop probe approach based on the nature of shared variables in the program. The eager writeback hardware technique proposed here will address the second condition and avoid the need to send snoop probes to all cores in the event of thread migration.

The OS may schedule threads to run on different cores across context switches to maximize resource utilization. However, one disadvantage of thread migration is that while moving to a new core the thread leaves behind information such as cache footprint, history in memory disambiguators, prefetchers, branch predictors, etc. Sometimes performance may be better off if core affinity [5, 33] is maintained as much as possible without conflicting with overall CPU utilization. The core affinity is not always possible, though. Therefore, it is necessary to snoop potentially modified lines when the OS migrates threads for the next time slice. This is one of the conditions that makes a snoopy-based cache coherence protocol to send snoop probes to all cores inevitable upon each access. To address this condition, we evict the modified contents from the lower level cache to the LLC just after the context switch. Normally, the context switch can be identified when important control registers representing the current process in the processor change. The same thread, while running on another core during the next time slice, will retrieve correct data from the last level cache without sending snoop probes to all the cores. The mechanism to flush modified lines to the last level cache is used in the modern processors while taking the cores to sleep state for saving power [6]. We expect the performance impact will be minimal, as many of these lines that belong to an old-time slice may get evicted after all because of conflict misses after the context switch. The performance impact resulting from eager writeback will be quantified later.

6. Essential Snoop Probe (ESP)

Although the SSP technique reduced the number of snoops, it did not completely eliminate the unnecessary snoop probes that returned clean responses because of aliases in the Bloom filters. We now present a compiler-assisted hardware technique called Essential Snoop Probe (ESP), a simple and complexity-effective mechanism that exploits the non-shared information in all types of the variables (stack, global, and heap) to reduce snoop probes down to the essential ones.

The ESP technique requires synergy between compiler and hardware to work effectively. The compiler, through various techniques explained further below, annotates the stack, global, and heap memory reference instructions with a Snoop-Me-Not (SMN) bit. This bit, when turned on, indicates that the accessed variables are not shared and snoops need not be sent to the other cores when processing this memory request. The hardware requires a small amount of logic to check the SMN bit annotation. Figure 9 shows the implementation of the ESP technique. As shown in Figure 9, when the SMN bit is set, a load access that reaches the LLC does not snoop the lower level caches of other cores, because the compiler explicitly indicated that this variable is not shared. On the other hand, if the SMN is not set, the hardware performs as usual (lines denoted by *esp*).

To address the self-modifying code condition, compiler needs to pass information if the program contains self-modified code to the microarchitecture. It is identified either from data flow analysis or from user defined *pragmas*. One approach is to use the compiler tool chain assign a non-zero value to a predefined reserved memory location. The predefined memory location can be in one of the sections that is part of the executable to indicate if the program contains self-modified code. The loader while loading the program reads this predefined memory location and sets the SMC control register (SMC-CR) bit during the program initialization phase before the program starts its execution. The microarchitecture checks this SMC-CR bit and prevents sending snoop probes to the I-cache when it is set. When the SMC-CR bit is set to zero to indicate that it is indeed executing a self-modified code, the microarchitecture

sends snoop probes to the I-cache. While executing self-modifying code, the SMC snoop probes (line denoted by *esp-smc*) to the I-cache are sent. The lines *esp* and *esp-smc* indicate the essential snoop probes.

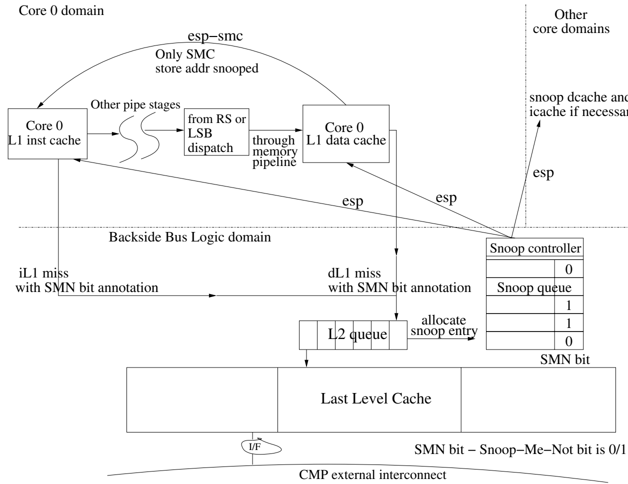

Figure 9. Essential Snoop Probe (ESP).

Compilers can use multitude of techniques to generate the SMN bit. Using the data flow analysis and algorithms, inter-procedure optimization, and other techniques, compilers can determine whether variables in a program are shared with other threads or programs. For example, variables explicitly declared as shared, or private in the OpenMP construct are supported by many compilers [35, 27]. To support parallel programming, the POSIX thread interface provides a better way of dealing with thread local storage (TLS) [2] using *__thread* C/C++ keyword. This new keyword allows thread-specific variables to be easily distinguished from others. In addition to compiler techniques, programming languages also provide *scope* for variables. The Java language has global scope for classes, package scope for fields and methods within a package, and procedure scope for local variables. Similarly, C/C++ language also provides storage scope. This scope information can also be used by the compilers to determine whether a variable is shared or not. Also, each thread has its own private stack to work with that is not visible to the outside world. Using a combination of techniques described above, compilers can determine if a variable in the program is shared or not. This gives the possibility of minimizing the number of snoop probes by not sending them to all the cores for reads and writes that reach the LLC. Whenever the compiler cannot determine for sure if a variable is shared or not, it leaves the SMN bit off. Also, the hardware will always send snoops when running the legacy code or when running the code generated by a compiler that does not support SMN bits.

7. Applicability of SSP and ESP Techniques to Large-Scale CMP (LCMP)

Large-scale system topologies like mesh, torus, and trees are built that involve tens or hundreds of processors using nodes and routers as a basis. Each node ranges from a single core to a multicore processor. The directory-based protocol is widely used for large-scale system design. We believe a hierarchical cache coherence protocol combined with hierarchical ring interconnects [3, 28] or concentrated mesh [8] design will be efficient for future large-scale CMP systems. In the hierarchical cache coherence protocol design, the snoopy-based cache coherence protocol that is suitable for fewer systems, can be used in an inner ring to connect a small group of neighboring cores together. The directory-based protocol can be used in the outer ring to connect inner rings together. In this kind of organization, the outer ring directory needs to keep track of inner rings that have the copy of data instead of individual cores in all rings, which gives better scalability at lower hardware directory cost. Our proposed Selective Snoop Probe (SSP) scheme can be used in the inner ring of large-scale CMP systems. On the other hand, the compiler-based Essential Snoop Probe (ESP) scheme is applicable to both small- and large-scale systems and is independent of the cache coherence protocol used. Also, snoop probe elimination resulting from self-modified code (SMC) is independent of the cache coherence protocol used, as the probes are within the core between data and instruction caches.

8. Experimental Results

We modified an x86 platform simulation infrastructure to evaluate the SSP and ESP techniques. The detailed cycle accurate simulator models a hypothetical future CMP system. The simulator executes the traces [11] collected from real-world applications including the external events such as DMA and interrupts. The traces are gathered for various categories: SPEC INT 2006, SPEC FP 2006, server, games and multimedia, and multi-threaded applications. In the multicore configuration, multiple copies of the same application are executed on each core, except for the multi-threaded category. The example applications in each category are shown in Table 2. Each category has ten applications, and each application has multiple traces that represent different characteristic portion of the application similar to the SimPoint [19] methodology. The traces are executed for 100 million instructions that cover many characteristic portions of the application after warming up all the caches, the TLBs, and the other hardware structures. The simulated configurations are two-core, four-core, eight-core, and 2x4-core (two processors, four-cores per processor). A total of 500 traces were executed and the simulated processor configuration is shown in Table 3. We used the CACTI 4.2 [1] 70nm model to determine the energy consumed by the data cache, instruction cache tag, and the hash arrays to evaluate the energy savings. The energy consumed by the 32KB cache is 0.4673 nJ, and the three 512-entry 10-bit Bloom filter is 0.0096 nJ. The leakage energy for 32KB cache is 0.1037 nJ, around 22.3% of the active energy.

Benchmark class	Example applications
Server	SpecJBB, TPCC
SPEC FP 2006	wrf, namd, lbm, soplex
SPEC INT 2006	hmmer, gobmk, omnetpp, gcc
Games and multi-media	shooters, realtime strategy, raytracer
multi-threaded applications	raytracer, cinebench

Table 2. Benchmark programs.

64-bit Processor Parameters	Values
Execution Engine	4-wide out-of-order
Reorder Buffer	256 entries
Load queue	96 entries
Store queue	64 entries
L1 TLB entries	128, 4 way
L1I cache	32KB, 8 way, 64B line, 4 cycles
L1D cache	32KB, 8 way, 64B line, 4 cycles
L2 cache	4MB, 16 way, 64B line, 8 cycles
Memory	2GB, DDR2 timings

Table 3. Processor model parameters.

Figure 10 shows the percentage of data cache energy savings per core using the SSP technique. It shows that in each core 5% to 10% of data cache energy savings in the 2C configuration and 30% to 65% in the 8C configuration are achieved. It also shows that the data cache energy savings increases with the number of cores on

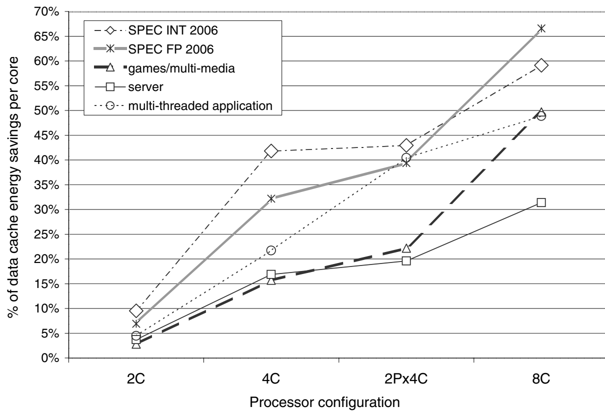

Figure 10. Energy savings in data cache per core using SSP.

the die, as the number of snoops to all the cores increases. The data cache energy savings in the 2Px4C dual-processor configuration is a little bit higher than the 4C configuration, because of the external snoop (snoop probes between CMP processors) filtering in the 2P case.

Figure 11 shows the percentage of tag energy savings in the instruction cache using the SSP technique. It shows that 50% to 70% of energy savings on average was achieved in the instruction cache tag across all processor configurations. The number of snoops to the instruction cache is determined by the number of writes and the RFOs as shown in Table 1. The major contributing factor to the instruction cache snoops are the store addresses in the program. As the percentage of store addresses across different programs do

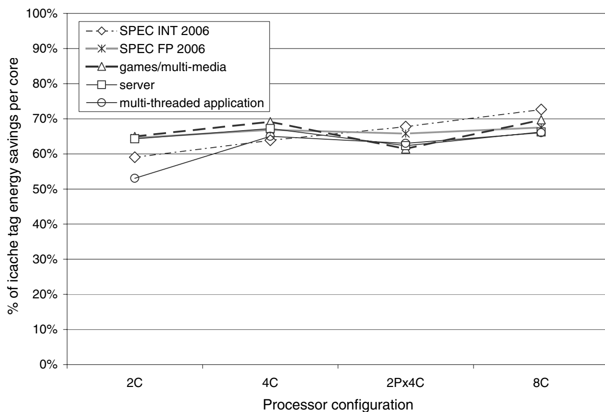

Figure 11. Energy savings in the instruction cache tag per core using SSP.

not vary much, the percentage of energy savings in the instruction cache tag also do not vary widely.

Figure 12 shows the performance impact using the SSP technique. It shows that on average there is nearly 1% to 2% performance improvement across various benchmark categories and different processor configurations. In one case, a performance improvement of 12% is achieved, as the reduced number of snoops to the lower level instruction and data caches provide more opportunities to service the regular instruction fetch, loads, and stores. Figure 13 shows the number of modified lines after the simulation completed. These modified lines are the ones that need to be flushed to the last level cache in order for the SSP technique to support the thread migration. We also consider the number of cycles required to flush these modified lines to the last level cache to support the eager writeback.

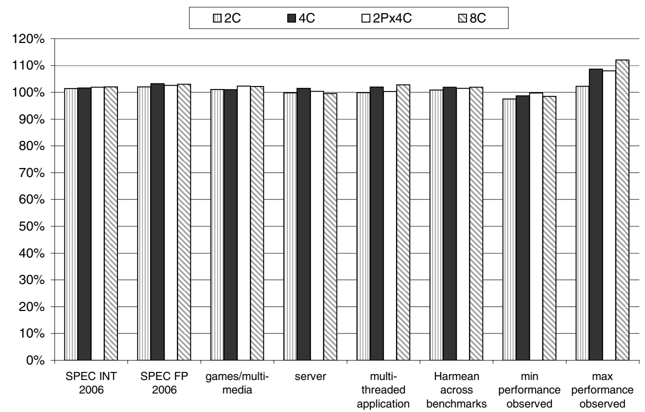

Figure 12. Performance impact using SSP.

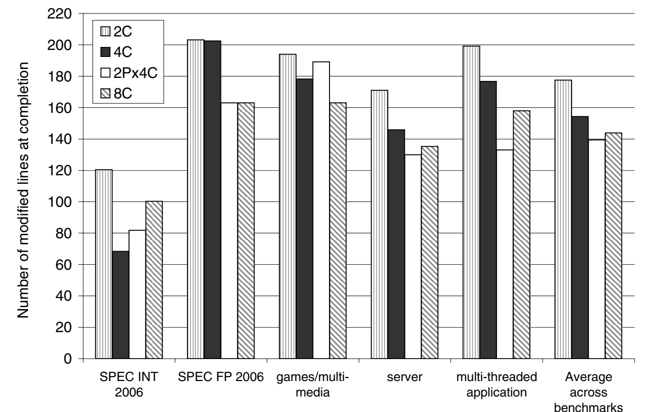

Figure 13. Number of modified lines after the program completion.

We implemented the RegionScout [25] technique proposed earlier by Moshovos *et al.* to compare with our SSP technique. We use a 64-entry fully-associative NSRT (Not Shared Region Table) cache, and a 512-entry CRH (Cached Region Hash) to record the regions that are locally cached with an 8KB region size. On average the SSP technique sends only 30-35% of snoop probes sent by RegionScout. We observed the following reasons for RegionScout to send more snoop probes than SSP technique: 1) the number of unique 8K regions are higher than the NSRT cache size creating many NSRT evictions 2) the SSP technique operates at the cache line granularity compared to the 8KB region granularity 3) all stack based accesses are eliminated in SSP unlike RegionScout where all the stack, global, and heap memory references are taken into account. Figure 14 compares the energy savings achieved using SSP and RegionScout techniques.

To evaluate the ESP technique, we show the potential energy savings that can be achieved using the hardware support that leverages the information generated by the compiler. The compiler can implement a variety of techniques to detect the sharing of the variables used in the program. Figure 15 shows the percentage of cache energy spent on the non-essential snoops. It shows that 5% (games category in dual-core configuration) to a maximum of 82% (SPEC FP 2006 in the 8-core configuration) were spent on the non-essential snoops that can be eliminated using the Essential Snoop Probe (ESP) technique. It also shows that the energy savings potential increases with the number of cores in the CMP, because of the increase in the number of non-essential snoop probes. Figure 15 shows that on average, 85% of instruction cache tag energy is spent

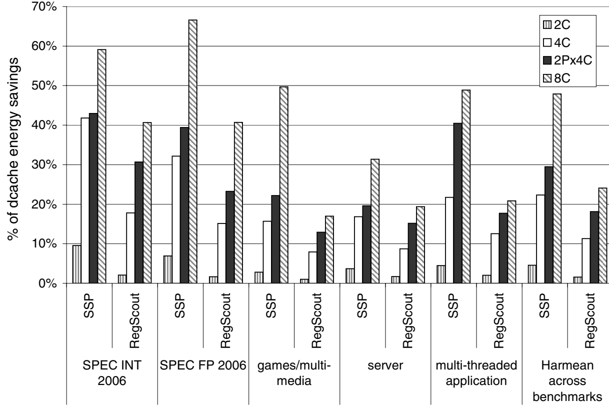

Figure 14. Energy savings comparison in data cache using SSP and RegionScout.

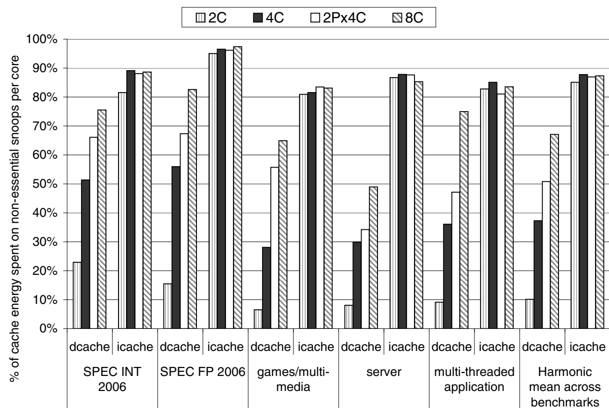

Figure 15. Energy savings in data cache and instruction cache tag per core using ESP.

on the non-essential SMC snoop probes. This percentage of savings does not vary much across benchmark suites and processor configurations. The ESP technique uses the synergy between the processor and the compiler to achieve higher energy savings compared to the SSP.

9. Related Work

It has been shown in prior work that many snoop requests miss all the remote caches in a shared-memory system [26, 16, 30, 12]. But there was not much insight given for these large misses. We found that one reason is that the stack accesses do not have global visibility and snoop probes will simply miss in remote nodes. Another reason is that many programs do not share any variable with the others running on the system. None of the earlier proposed hardware or software solutions were cognizant of the semantics of the variables used in the program. A key difference between our ESP technique and the techniques proposed by others is that the ESP reduces the number of snoop probes to the essential ones to maintain cache coherency for the entire execution of the program rather than depend on the spatial or temporal locality of memory references. Also, both our SSP/ESP techniques take self-modifying and cross-modifying code into account.

Previous work on snoop energy reduction relied on blocks of memory to optimize the snoops. For example, Ekaman *et al.* [16] proposed a page sharing table that uses vectors to identify the sharing at page level. Saldanha and Lipasti [30] proposed serial snooping to reduce the energy in shared-memory multiprocessors. In Include Jetty [26], each node avoids the snoop accessing the L2 cache by first checking the Jetty structure. One variant (Exclude Jetty) used the temporal and spatial locality of shared data by caching the recently missed snoops. The Jetty techniques, in their original form, were designed to handle external snoops in SMPs. As shown in [15], they do not work as effectively when applied to CMPs: Include Jetty does not prevent a majority of unnecessary snoops and Exclude Jetty requires prohibitively large hardware. Our SSP and ESP techniques work well in a CMP as they selectively send snoop probes where needed.

Moshovos proposed RegionScout [25] where each node can determine in advance if a request would miss in all other nodes based on region sharing information. When a node requests a block in a region marked non-shared, there is no need to probe any other node. The key difference of RegionScout from ours is that both the SSP and ESP techniques selectively send the snoop probes or completely eliminate them if possible, whereas RegionScout has to broadcast the initial request to identify Region Hit information. The RegionScout technique requires bus interconnect architectures as a wired-OR signal is needed to notify a region hit. In contrast, our techniques are not limited by the choice of interconnect architecture used in a CMP system.

Cantin *et al.* [12] proposed a technique to reduce the number of broadcast snoops required to maintain coherency in an SMP system. Their idea is similar to RegionScout, and requires a hardware structure called *Region Coherency Array* as well as extra bits in the processor interconnect. Our SSP technique is different because it completely eliminates snoops for stack accesses and selectively dispatches snoops for non-stack accesses. Our ESP differs from RegionScout and Region Coherency Array as it uses compiler's support to reduce snoop probes to the essential ones based on the semantics and sharing properties of the variables used in the program without any storage based hardware structures.

Recently, Dash *et al.* [14] proposed an application-driven snoop filtering mechanism for the embedded systems. In their technique, the information regarding the shared array is made available to the OS. The OS tags the TLB entries with a region id that denotes if the page is shared or not. On a last-level cache miss, the address on the shared bus is tagged with a three bit region id. This region id is used by the snoop controller in each processor to filter the snoops to the lower level caches.

10. Conclusion

In this paper, we proposed and evaluated two novel snoop filtering mechanisms based on the access semantics of the variables used in programs. Given the fact that the stack variables are all private, we modified the existing snooping coherence protocol with minimal hardware support to relax the generic and conservative implementation which indiscriminately broadcast snoop probes to all cores. First, we proposed a hardware-only technique called Selective Snoop Probe (SSP) to eliminate all the snoop probes for stack accesses. In addition, counting Bloom filters were employed based on MESI state transitions to further reduce the number of snoop probes caused by non-shared and non-stack accesses. This hardware-only technique reduced the number of snoops substantially, however, not all the unnecessary snoops can be eliminated at runtime due to the aliasing effects of Bloom filters. To address this limitation, a compiler-assisted technique called Essential Snoop Probe (ESP) was proposed to include all variables in the program by annotating the instructions with a Snoop-Me-Not (SMN) bit set by the compilers representing the need to snoop during execution.

The advantage of using the SSP is that all the functionality is implemented in the hardware and transparent to the programmers. In addition, previously compiled binaries can benefit from this tech-

nique without recompilation. However, as there is no information provided by the software, the energy savings achieved is limited using the SSP technique. On the other hand, the ESP technique lets the compiler take full advantage of the hardware support to achieve higher energy savings. We showed that the SSP technique saved 5% to 65% of data cache energy, and 50% to 70% of instruction cache tag energy per core across different processor configurations with 1% to 2% performance improvement. We also showed that nearly 5% to 82% of data cache energy and 85% of instruction cache tag energy in each core was spent on the non-essential snoop probes that can potentially be saved using the compiler guided ESP technique.

Our techniques can easily be extended to optimize future integrated platforms such as AMD Fusion. In such systems, we expect that the number of snoops initiated from the graphics core to the rest of the cores on-die will be much higher given the nature of graphics workloads. Our proposed techniques when used by CUDA compilers can provide effective guidance to the hardware to minimize snoop traffic, substantially reducing the requirement of bus bandwidth and improving the overall power and performance of such heterogeneous systems.

Acknowledgments

This research was supported in part by NSF Grant CNS-0325536 and an NSF CAREER Award CNS-0644096.

References

[1] CACTI 4.2. In *http://quid.hpl.hp.com:9081/cacti*.

[2] ELF handling for Thread Local Storage. In *people.redhat.com/drepper/tls.pdf*.

[3] IDF 2006: Terascale Processing Brings 80 Cores to your Desktop. In *http://www.pcper.com/article.php?aid=302&type=expert&pid=3*.

[4] Microprocessor cache-coherency snooping. In *http://www.warthman.com/ex-inqr.htm*.

[5] Performance guidelines for AMD Athlon 64 and AMD Opteron. In *www.amd.com/us-en/assets/content-type/white-papers-and-tech-docs/40555.pdf*.

[6] Power and Thermal Management in the Intel Core Duo. In *www.intel.com/technology/itj/2006/volume10issue02/art03-Power-and-Thermal-Management/p03-power.htm*.

[7] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *Proc. of Int'l Symp. on Computer Architecture*, 1988.

[8] J. Balfour and W. J. Dally. Design tradeoffs for tiled cmp on-chip networks. In *Proc. of Int'l Conf. on Supercomputing*, 2006.

[9] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single Chip Multiprocessing. In *Proc. of Int'l Symp. on Computer Architecture*, 2000.

[10] B. H. Bloom. Space/time Trade-offs in Hash Coding with Allowable Errors. *Communication of the ACM 1970*.

[11] D. Boggs, A. Baktha, J. Hawkins, D. T. Marr, J. A. Miller, P. Roussel, R. Singhal, B. Toll, and K. S. Venkatraman. Performance analysis and validation of the Intel Pentium 4 processor on 90nm Technology. *Intel Technology Journal*, 8(1), 2004.

[12] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving multiprocessor performance with coarse-grain coherence tracking. In *Proc. of Int'l Symp. on Computer Architecture*, 2005.

[13] E. Dahlen, J. Gustin, S. Meredith, and D. Moran. The 82460GX Sever/Workstation Chip Set. *IEEE Micro*, 2000.

[14] A. Dash and P. Petrov. Energy-efficient cache coherence for embedded multi-processor systems through application-driven snoop filtering. In *EUROMICRO DSD 2006*.

[15] M. Ekman, F. Dahlgren, and P. Stenstrom. Evaluation of Snoop Energy-Reduction techniques for Chip-Multiprocessors. *Workshop on Duplicating, Deconstructing and Debunking in conjunction with ISCA 2002*.

[16] M. Ekman, P. Stenstrom, and F. Dahlgren. TLB and Snoop Energy-reduction using Virtual Caches in Low-power Chip Multiprocessors. In *ISLPED 2002*.

[17] M. Ghosh, E. Ozer, S. Biles, and H.-H. S. Lee. Efficient System-on-Chip Energy Measurement with a Segmented Bloom Filter. In *ARCS 2006*.

[18] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-Fault Recovery for Chip Multiprocessors. In *Proc. of Int'l Symp. on Computer Architecture*, 2003.

[19] G. Hamerly, E. Perelman, J. Lau, and B. Calder. SimPoint 3.0: faster and more flexible program analysis. *Journal of Instruction Level Parallelism 2005*.

[20] H.-H. S. Lee and C. S. Ballapuram. Energy efficient D-TLB and Data Cache using Semantic-aware Multilateral Partitioning. In *Proc. of Int'l Symp. on Low-Power Electronics and Design*, 2003.

[21] H.-H. S. Lee, M. Smelyanskiy, G. S. Tyson, and C. J. Newburn. Stack Value File: Custom Microarchitecture for the Stack. In *Proc. of Int'l Conf. on High Performance Cmputer Architecture*, 2001.

[22] H.-H. S. Lee and G. S. Tyson. Region-Based Caching: An Energy-Delay Efficient Memory Architecture for Embedded Processors. In *Proc. of Int'l Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, 2000.

[23] D. Marr, S. Thakkar, and R. Zucker. Multiprocessor validation of the Pentium Pro microprocessor. *COMPCON 1996*.

[24] A. Mendelson, J. Mandelblat, S. Gochman, A. Shemer, R. Chabukswar, E. Niemeyer, and A. Kumar. CMP Implementation in Systems Based on the Core Duo. *Intel Technology Journal*, 10(2), 2006.

[25] A. Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. In *ISCA 2005*.

[26] A. Moshovos, G. Memik, B. Falsafi, and A. N. Choudhary. JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers. In *Proc. of Int'l Conf. on High Performance Cmputer Architecture*, 2001.

[27] D. Novillo. OpenMP and Automatic Parallelization in GCC. In *GCC developers summit*, 2006.

[28] G. Ravindran and M. Stumm. A performance comparison of hierarchical ring- and mesh- connected multiprocessor networks. In *Proc. of Int'l Conf. on High Performance Cmputer Architecture*, 1997.

[29] A. Roth. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization. In *Proc. of Int'l Symp. on Computer Architecture*, 2005.

[30] C. Saldanha and M. Lipasti. Power efficient cache coherence. *Workshop on Memory Performance Issuses in conjunction with ISCA 2001*.

[31] S. Sethumadhavan, F. Roesner, J. S. Emer, D. Burger, and S. W. Keckler. Late-binding: Enabling Unordered Load-store Queues. In *Proc. of Int'l Symp. on Computer Architecture*, 2007.

[32] W. Shi, H.-H. S. Lee, L. Falk, and M. Ghosh. An Integrated Framework for Dependable and Revivable Architectures Using Multicore Processors. In *Proc. of Int'l Symp. on Computer Architecture*, 2006.

[33] M. S. Squillante and E. D. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 1993.

[34] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving Both Performance and Fault-Tolerance. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2000.

[35] X. Tian, A. Bik, M. Girkar, P. Grey, H. Saito, and E. Su. Intel OpenMP C++/Fortran Compiler for Hyper-Threading Technology. *Intel technology Journal*, 3(1), 2002.