Snoop Optimization Techniques For Chip-Multiprocessors

ABSTRACT

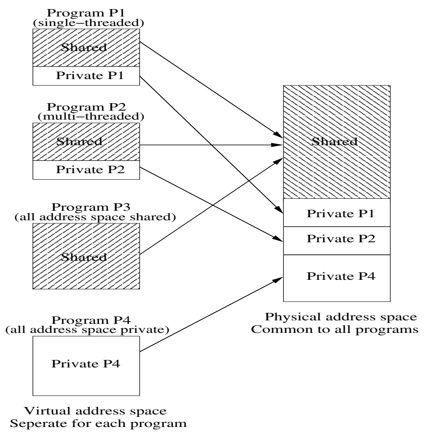
Increasing the number of cores per die in every generation of process technology is the current trend in the industry. Most of the current microarchitecture research use chip-multiprocessors (CMP) as the baseline to analyze problems. One of the problems facing CMP systems is the growing number of snoops necessary to maintain cache coherency and support self/cross-modifying code leading to power and performance-limiting events. In this work, we analyze the internal and external snoop behavior in a CMP system based on the semantics and sharing of the variables used in the program to relax the conservative snoopy cache coherence protocol. Based on the observations and analyses, we present two novel techniques called Selective Snoop Probes (SSP) and Essential Snoop Probes (ESP) to reduce power without compromising performance. Our simulation results show that using SSP technique 5% - 65% data cache and 50% - 70% instruction cache energy savings in different processor configurations can be achieved with 1% - 2% performance improvement. We also show that 5% to 82% of data cache and 85% of instruction cache energy is spent on the non-essential snoop probes that can be saved using the ESP technique.

1. INTRODUCTION

Continuous miniaturization of devices in process technology has brought on-die homogeneous and heterogeneous multi-core processors or chip multiprocessors (CMP) into all market segments ranging from servers to mobile products. In addition to improving performance, CMP can also be used for addressing emerging issues such as security [46], reliability [31, 45], etc. In a CMP system, maintaining cache coherence is a complex task and the support for self-modifying code (SMC) and cross-modifying code (XMC) further increases the design complexity. There are two kinds of snoops, internal and external in a CMP-SMP systems (CMP based SMP systems). The *internal snoops* are triggered and responded by the cores *within* the same CMP, while the *external snoops* are triggered and responded by *different* CMPs in a CMP-SMP system. Prior research work have dealt with the analyses and optimization of external snoops in a SMP environment, but have limited analyses in internal snoop behavior comprising self-modifying and cross-modifying code snoops in a CMP. The snoop response time is becoming critical in a CMP system due to the following reasons: 1) increasing number of cores per die 2) the continuous extension of the vertical cache hierarchy 3) the increasing size of cache capacity and load/store buffers leading to timing critical circuit paths, that provide snoop responses back to the requesting node. This 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 the snooping requirements manifest themselves differently in CMP systems having different cache inclusion policies and leading to different response times.

Cache inclusion properties were first studied by Baer and Wang [35] to provide design guidelines for cache hierarchies and facilitate the cache coherence implementation. Strongly inclusive caches are generally used in academic research while commercial processors implement different inclusion policies. For example, IBM's Power5 [17] uses strongly inclusive caches [35], the Intel Pentium Pro uses weakly-inclusive caches [25], the COMPAQ Piranha [36] and AMD Athlon [3] use exclusive caches in their design. In exclusive caches, the data is in only one of the caches in the hierarchy to increase the effective cache capacity. All three cache policies have their pros and cons. In both the 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 that are nearer to the core) and other related memory buffers in a CMP. This snooping-all technique may not scale well with the increasing number of cores as it increases power and inter-core communication. In the strongly inclusive case, the snoops probe all the lower level caches only when the cache line is present in the last level cache in order to obtain the most recent version from the core that owns it. One alternative to avoid sending snoop probes to all the cores when using inclusive caches is to add a *shadow* set of all lower level cache tags near the shared last level cache to maintain coherence. Unfortunately, the overhead of maintaining these shadow tags will become higher as the number of cores, the cache hierarchy, and the size of the lower level caches increase.

The snoopy cache coherence protocol and its implementation in general is conservative. It always assumes that all the variables used in a program are shared among the threads or programs that are running in the system. In other words, to maintain functional correctness, all the reads and writes that reach the last level cache must send snoop probes to the other cores or processors in the sytem. However, snooping for all program variables in all the cores can sometimes be avoided if we know apriori that certain variables need not be snooped based on their properties. The user input, the programming languages used to design an application, and the differentiation between single and multi-threaded application can play an important role in determining the number of snoop probes generated and their behavior. By efficiently utilizing the programming language constructs and improving the contract between the user, programming language, and the application, snoops can be optimized. The reason for the cache coherent snooping protocol and its implementation to be conservative is twofold: 1) cache coherence protocol does not distinguish between the variables that are shared and the ones that are not 2) when a thread migrates from one core to another due to OS scheduling, it typically leaves behind its old work (i.e. modified variables) in the old caches and is required to snoop all the cores/processors to continue its work in the new core. In this paper we try to address these drawbacks with hardware and software based approaches. Our goal is to relax the conservative nature of the cache coherent protocol and its implementation to not snoop for all the program variables in all the cores and thereby reduce the excessive bandwidth and resources they use.

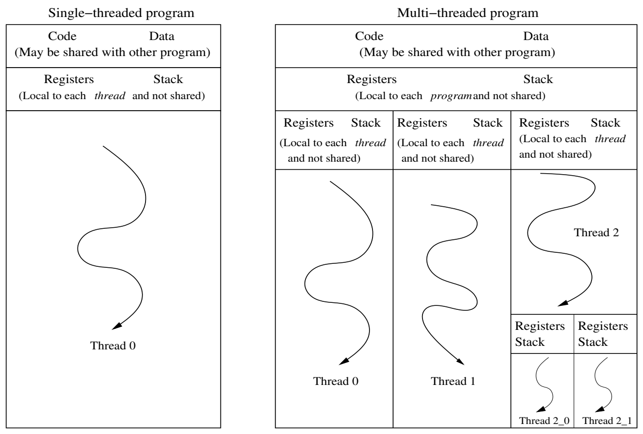


Program P1 (single−threaded)
Shared
Private P1
Program P2 (multi−threaded)
Shared
Private P2
Program P3 (all address space shared)
Shared
Program P4 (all address space private)
Private P4
Virtual address space
Seperate for each program
Shared
Private P1
Private P2
Private P4
Physical address space
Common to all programs

Figure 1: Various program categories

The rest of this paper is organized as follows. Section 2 discusses different types of programs and the semantics of the variables. In Section 3 we describe different types of snoops, the snoop flow in a typical CMP machine and the different hardware structures that need to be snooped based on the memory request. Section 4 describes the Selective Snoop Probe (SSP) hardware snoop optimization. Section 5 describes a method to handle snoops in the event of core migration. Section 6 describes the Essential Snoop Probe (ESP) hardware/software mechanism. Section 7 explains the details of the traces used, simulation methodology, and the analysis of experimental results. Section 8 discusses prior research on snoop energy reduction techniques and the differences between our work from these prior efforts. Finally, Section 9 concludes the paper by comparing our novel SSP and ESP techniques.

2. PROGRAM VARIABLES AND SNOOP PROBES



Single−threaded program
Code Data
(May be shared with other program)
Registers Stack
(Local to each thread and not shared)
Thread 0
Multi−threaded program
Code Data
(May be shared with other program)
Registers Stack
(Local to each program and not shared)
Registers Stack
(Local to each thread and not shared)
Thread 0
Thread 1
Thread 2
Registers Stack
Thread 2_0
Thread 2_1

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

Figure 1 illustrates four different program categories based on the nature of data shared in the program. Programs P1 and P2 are similar and contain both shared and private variables. The difference between them is that while P1 is single-threaded and shares its global variables with other *programs* in the system, P2 is multi-threaded and shares its variables with other *threads and other programs* in the system. The third category is represented by program P3, where all the variables are assumed to be shared with other programs, which is normally the case assumed by the 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 P1 and P2. In general, both code and data are assumed to be shared with other programs in the system. This is the region marked *shared* in Figure 1. On the other hand, the registers and the stack are all local to each thread and are not globally visible to other programs. The variables in this region are shown as *private* for P1, P2, and P4 in Figure 1 which are not visible to the outside world.

The knowledge about the variables that are shared in a program provides a very good opportunity to optimize snoops in a shared memory system environment. Current snoopy 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 such as semaphores or locks. This information can potentially be identified during the program compilation. In this case the program might be "self-sufficient" in the cache coherent sense, where the reads and writes need not probe other cores or processors in the system. Similarly, the programmer can give his/her input or it can be identified during compilation that the program does not contain self-modifying code. These inputs are valuable to the underlying processor to reduce the indiscriminate snoop probes that are sent in order to improve the overall system power and performance.

In this paper we propose a hardware only technique called Selective Snoop Probe (SSP) that uses the properties of the stack variables to filter unnecessary snoops, and a hybrid hardware/software technique called Essential Snoop Probes (ESP) 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 these techniques can be adopted in all types of inclusive/exclusive cache policies.

3. SNOOP CLASSIFICATION

In a CMP where several cores on the same die share the last level cache, internal snoops to the lower level caches are inevitable; they are not only necessary to maintain cache coherency but also to support self-modified code (SMC) and cross-modified code (XMC). In this section, snoops due to 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) is a piece of code that changes its own instructions while it is executing by writing to them. Many commercial processors [10, 4, 11, 1] provide support for self/cross-modifying code. There are many applications that use this feature, and one example is Just-In-Time compilers that use SMC to generate optimized code amid 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 in the prefetch buffers. For example in the Pentium 4 processor, if a write in a code segment matches the target instruction that is already decoded and resident in the trace cache, the entire trace cache will be invalidated. To support this behavior, each store address needs to send a snoop probe to the instruction cache. If found, the instruction cache needs to

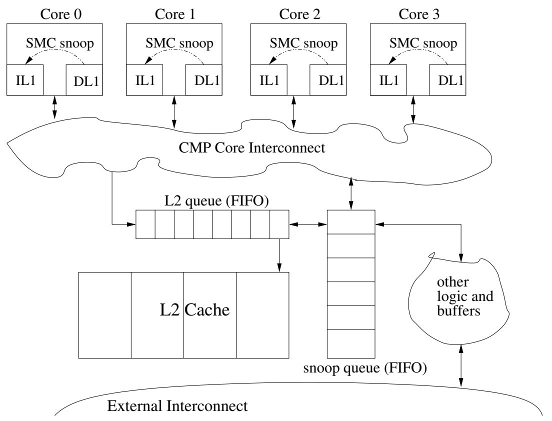


Figure 3: Snoop flows in a Quad-core CMP system

invalidate the corresponding cache line. On every code access to the last level cache, a snoop probe to the same core's data cache and store buffers are sent to support SMC. In addition, snoop probes to the other cores in the CMP 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 split private L1 instruction (iL1) and data caches (dL1), and all four cores share one unified last level L2 cache (sL2). The L2 is accessed on any L1 instruction or L1 data cache miss by all four cores, L1 and L2 prefetchers, and the external snoops in a MP system. The CMP interconnect that connects all the lower level cores can be bus, ring, or arbiter based interconnect. The requests that need to access the L2 cache are queued in a common hardware structure called L2 queue and the L2 access is pipelined. The internal and external snoop requests are queued in another hardware structure called snoop queue [6, 30]. 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' L1 instruction and data caches, load/store buffers, MSHR (Miss Status Handler Registers), based on the type of memory request. It is important to note that each snoop request allocated in the snoop queue spawns multiple snoop probes to different hardware structures in all the cores. The snoop response, and data if necessary, are propagated to the requesting cores. It is also necessary to do a snoop queue match before sending the responses to the external bus, and before writing a cache line to the last level cache for the requests reaching the last level cache to maintain the cache coherency and memory consistency. Thus, it becomes crucial to reduce the occupancy of the snoop queue to improve performance.

Snoop flows differ based on the type of the CMP interconnect architecture that is used. The interconnect architecture for CMP is an active area of research [26, 40, 38, 18] and there are many possible implementations. In a ring interconnect, the snoops are sent across the ring and all the requests to the cores are queued and processed. The cores return the snoop response, and data if applicable, over the ring. The snoops can consume large amount of bandwidth if not properly handled or optimized. Irrespective of the kind of implementation, it is important to complete the snoop probes and return the responses to the requesting core as quickly as possible or to avoid the snoop probes completely whenever possible in order to improve the overall system performance.

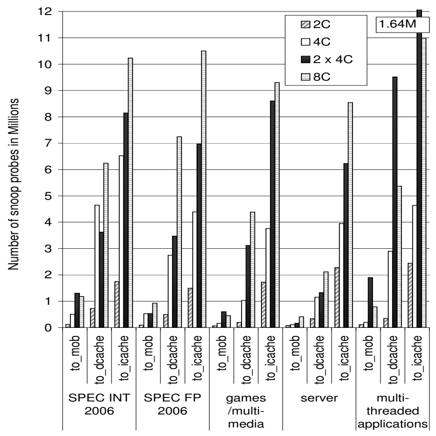


Figure 4: Snoop probes in different benchmarks

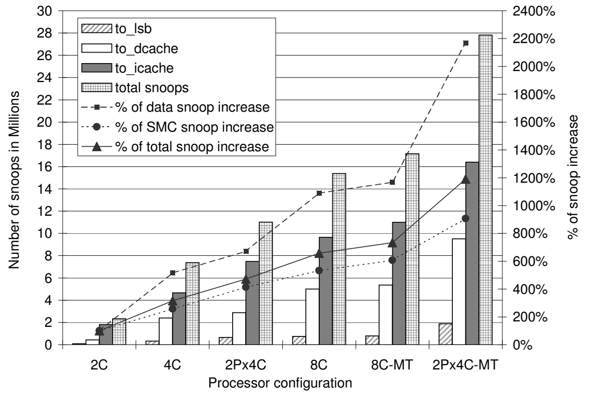


Figure 5: Snoop probes and snoop rate in different processor configuration

3.3 Snoop triggers and snoop probes

Table 1 shows the internal snoop trigger points and the hardware structures that need to be snooped to support cache coherency and S/XMC snoops. The first column shows the incoming requests to the last level cache, in this case the L2. The next two columns correspond to the hardware structures belonging to 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 RFOs (request-for-ownership) need to probe at least 3 to 4 (the number of relevant hardware structures) times the number of cores in the CMP to maintain cache coherency. A code fetch that reaches the last level cache is much more expensive than data read or RFO (request-for-ownership) as it sends snoop to all the modules shown in the snoop table, but is mitigated by the lower instruction cache miss rates. This table clearly shows that as the number of cores per die and the vertical cache hierarchy grows, these internal snoops will likely become the bottleneck in limiting the performance and power.

Figure 4 and Figure 5 show the number of snoop probes received by each hardware structure for various benchmark categories in different processor configurations.[1] Each core

[1]The graph shows the results collected from around 200 traces for 6M instructions each based on the simulation

Incoming events to the last level cache	**iL1 of this core**	**dL1 of this core**	**LSB of this core**	**dL1 MSHR, WBB of this core**	**iL1 of other 3 cores**	**dL1 of other 3 cores**	**LSB of other 3 cores**	**dL1 MSHR, WBB of 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	-	-	XMC snoop to invalidate line	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	snoop SMC
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

in all configurations has split L1 instruction (iL1) and L1 data (dL1) caches. All cores in one processor share the same L2 (sL2). Also note that the L2 employs a weakly-inclusive policy, where a cache line in L1 may not be in L2. Figure 4 shows that the number of snoops to three microarchitectural modules — the Load-Store Buffer (LSB), the L1 data cache, and the L1 instruction cache. The figure illustrates that the number of snoops to the instruction cache 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. The 2C, 4C, and 8C configurations represent 2, 4, and 8-core CMP systems respectively on which 2, 4, and 8 copies of a single-threaded program is run. The 2Px4C configuration is a system with two processors, each processor comprising of four cores where each core is running one copy of a single-thread application. The 2Px4C-MT contains 2 quad-core processors running an 8-way multi-threaded application while the 8C-MT is simply an oct-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 slightly more number of snoops than its single-thread counterpart as the shared variables between threads increase the snoop probes. Also, there is a slight increase in the snoop traffic due to the external snoops in the dual-processor configuration 2Px4C compared to the uni-processor configuration 4C. And there is a high increase in number of snoops when mutli-threaded workload is run on dual-processor (2Px4C-MT) configuration due to shared variables and external snoops. The secondary axis of Figure 5 shows the percentage snoop increase with respect to 2C configuration as the baseline. Although both Figure 4 and Figure 5 show that the number of snoops to the instruction cache is higher, 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 core increases beyond 8 or 16 cores, the snoops to the data caches and the Load-Store Buffer (LSB) structures will limit the performance. The number of snoops tend to increase in multi-threaded applications and is further aggravated in a MP environment, thereby limiting the performance improvement that is 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 in the program and the nature of the application is not conveyed by the program to the processor. To address these issues, we propose 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 the 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.

methodology described in the experimental section.

4. SELECTIVE SNOOP PROBES (SSP)

In this section, we describe and discuss our hardware solution that selectively filters snoop probes for the requests reaching the last level cache. Each access to the last level cache spawns snoop probes as described in Table 1 to obtain the most up-to-date copy of the data. The requests that reach the last level cache can be divided into two types based on the region of memory they access: stack region accesses and non-stack region accesses. The reason we partition accesses into these two types is based on a simple observation — the snoop probes generated due to the stack access requests should always receive a clean response from the other cores as the stack memory region is considered private to its own thread running on a core as described in Section 2. The snoop probes generated by the 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 the requests that access the stack region and those that receive a clean response by the non-stack accesses are candidates that can be removed. We observed for all benchmarks that the number of positive (hit or modified) responses from the cores, in fact, are much lesser than the number of the snoop probes sent. The main reason for receiving clean responses is twofold: 1) the snoop probes to the local thread stack variables are clean as it is invisible outside its own core and is not shared by any other process 2) programs typically do not contain self-modifying codes and are single-threaded without sharing variables with others. Unfortunately, modern day architectures do not distinguish these types of vari-

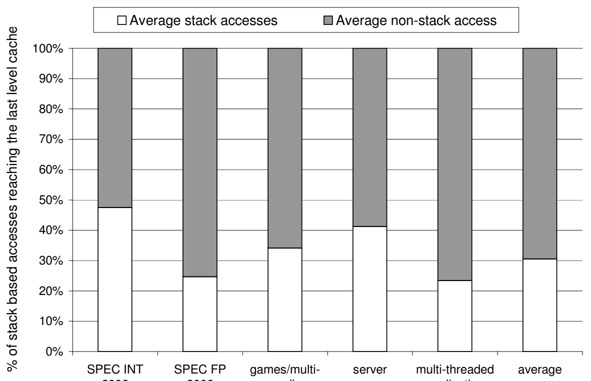


Figure 6: Stack accesses to the last level cache

ables, resulting in a large number of unnecessary snoops. These snoops and their clean responses consume power in communicating the tranactions, snooping hardware structures, and wasted bandwidth. It can also affect the 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 stack-bit (S-bit) annotation to eliminate snoop probes caused by the stack accesess. Meanwhile, MESI-state based counting Bloom filters are proposed to eliminate the snoop probes caused by the non-stack accesses that give clean responses.

Another interesting behavior exhibited by programs is that stack accesses typically account for a significant portion of all the memory references [33]. We profiled several benchmark suites to determine the distribution of memory accesses that reach the last level cache. Figure 6 shows the results of stack and non-stack access distribution. On average, about 30% of memory references go to the stack using stack and frame pointers as the source or destination registers. As shown in [34], the dominant method to access stack memory is via the stack pointer register and/or frame pointer register in an x86 architecture although other means such as via an offset from a general purpose register is not completely prohibited but rather uncommon. By decoding the source or destination register identifier, processor can isolate memory instructions that go to the stack region. To enable this isolation in the hardware, we propose to annoate load and store instructions with a single bit in the decode stage to indicate whether an access is going to the stack. Each request reaching the last level cache carry this annotation as part of the operation.

Figure 7 shows the details of internal and external snoop filtering mechanism based on the programming semantics of the variables. 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. Bloom filters [23] have been widely used in various microarchitectures for optimizing performance and power [21, 43, 28, 19, 41]. It 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, two distinct counting Bloom filters labeled ① and ② shown in Figure 7 were added to each core — one to track the valid cache lines in the instruction cache and eliminate the unnecessary SMC snoops, and the other one to track the data cache lines and eliminate those snoops that receive clean responses from non-stack accesses. The updates to the counting Bloom filter are denoted by the lines marked u1 and u2, and the reads are denoted by lines marked r1 and r2 shown in Figure 7. The operation of SSP is divided into three main parts: SSP for SMC, SSP for stack region accesses, and SSP for non-stack region accesses.

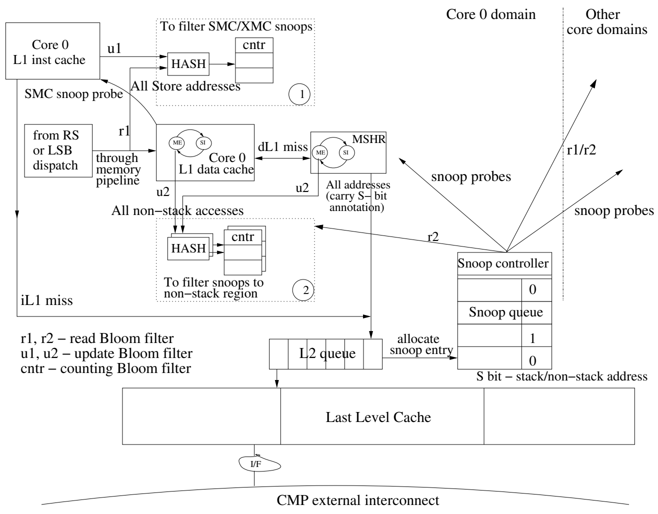


Figure 7: Selective Snoop Probes (SSP)

4.1 Selective snoop probes for SMC (SSP-SMC)

The counting Bloom filter in the box labeled ① in Figure 7 tracks all the valid lines in the instruction cache. Whenever an instruction cache line becomes invalid, it is removed from this Bloom filter. The addition and removal to the Bloom filter is denoted by the line u1. On a store address dispatch by the RS (Reservation Station) or LSB (Load-Store Buffer), the data cache control unit first looks up this Bloom filter (denoted by r1). The SMC snoop probe to the instruction cache is sent only if a lookup to the Bloom filter generates a hit. Thus unnecessary SMC snoop probes to the instruction cache are eliminated.

4.2 Selective snoop probes for stack accesses (SSP-SR)

The requests that reach the last level cache carry the 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. Before spawning the snoop probes, the snoop controller looks at the S-bit to determine if it is necessary to do so. If the S-bit is set, it does not send the snoop probes and thus they are eliminated. The stack access requests do not use any Bloom filter and use only the S-bit annotation for its operation. Around 30% of the accesses that reach the last level cache shown in Figure 6 fall under this category.

4.3 Selective snoop probes for non-stack accesses (SSP-NSR)

On average the remaining 70% of the requests reaching the last level cache go to the non-stack region. The counting Bloom filter in the box labeled ② in Figure 7 tracks all the non-stack accesses that look up the data cache and the MSHR as denoted by the lines marked u2. The snoop controller looks up the counting Bloom filters based on Table 1 and gathers information before spawing snoop probes (denoted by the lines r2 and r1/r2). This information is ob-

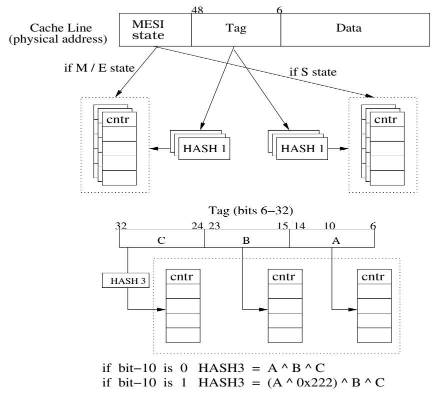


Figure 8: Hash functions used in counting Bloom filters

tained only for the non-stack accesses. The snoop probes to the instruction cache 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 the Bloom filter can miss those lines due to the aliases resulting from 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 the counting Bloom filter, the effectiveness of the hash functions, and the advantages of using our SSP technique are discussed in the following sections.

4.4 Bloom filters and hash functions

The counting Bloom filter in box ② of Figure 7 records the non-stack accesses based on the state transition of MESI protocol.[2] 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 the S(hared) state. If the MESI state of a cache line is transitioned to M/E state, its signature is recorded by the ME-Bloom filter. Similarly, if 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 the Bloom filters is to reduce aliasing and thereby decreasing the number of false-positives. This is based on the observation from all benchmarks that more snoops take cache lines to the S state than to the M or E state as load instructions are exectued more frequently than stores.

The hash functions for the counting Bloom filters in boxes ① and ② use the 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.[3] The first array is indexed directly by the lower order bits [14:6]. The second array is indexed by bits [23:15] of the address. The third array is indexed by XOR-ing the bits [14:6], [23:15], and bits [32:24] if bit 10 is 0. Otherwise, instead of directly using the bits [14:6], bits [14:6] is XOR-ed with 0x222 as shown in the Figure 8. This combination of hashing is done in order to distribute the indexing of the addresses to all the entries to reduce aliases.

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

[2]The invalidation-based MESI cache coherence protocol was used in this work.

[3]Note that the hash functions are fixed. We did not use different hashes for different benchmark programs based on profiling in our experiments.

5. EAGER WRITEBACK TO LAST LEVEL CACHE

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

The OS may schedule threads to run on different cores across context switches to maximize the overall CPU 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 the performance may be better off to have core affinity [15, 44] as much as possible without conflicting with the overall CPU utilization. The core affinity is not always possible, though. Therefore, when the OS migrates a thread for the next time slice, it is necessary to snoop for obtaining the potentially modified lines. This is one of the conditions in a snooping cache protocol that makes the snooping of all cores inevitable upon each access. To address this condition, we flush the contents of the modified lines from the lower level cache to the last level cache just before the context switch. Normally, the context switch can be identified when important control registers [9] representing the current process in the processor changes. As such , next time when the same thread running on another core will retrieve the correct data from the last level cache without snooping all the cores. The mechanism to flush the modified lines to the last level cache is used in the modern processors while taking the cores to sleep state in order to save power [16, 2]. We expect the performance impact will be minimal as many of these lines belonging to the old time slice may get evicted due to conflict misses after the context switch anyways. The performance impact due to eager writeback will be quantified later.

6. ESSENTIAL SNOOP PROBES (ESP)

The Selective Snoop Probe (SSP) technique exploited the knowledge about the semantics of the variables used in the program, specifically the stack variables using the S-bit annotation to reduce the unnecessary snoop probes.

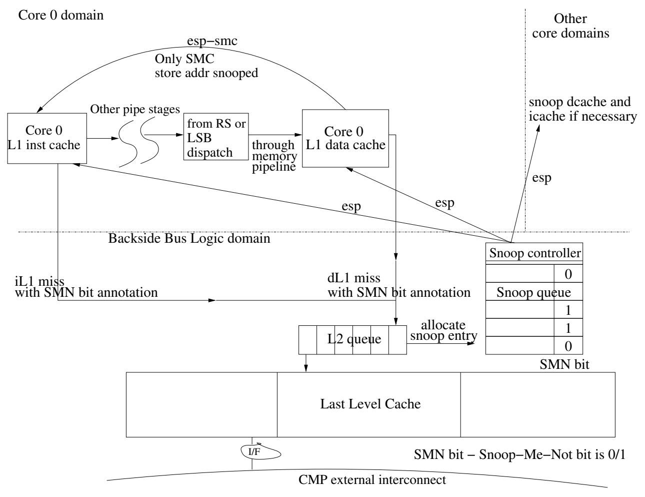


Figure 9: Essential Snoop Probes (ESP)

By adding counting Bloom filters, the non-shared global and heap variables were identified. Although the SSP technique reduced the number of snoops, it did not completely eliminate the unnecessary snoop probes that returned clean responses as Bloom filters had aliasing effects. We now present a compiler-assisted hardware technique called Essential Snoop Probe (ESP). The ESP technique is a simple and complexity-effective mechanism that exploits the non-shared information in all types (stack, global, and heap) of variables used in the program to reduce the snoop probes to the essential ones.

The Essential Snoop Probe (ESP) technique requires synergy between the compiler and the hardware to work effectively. The compiler, through various techniques explained further below, annotates memory instructions with a Snoop-Me-Not (SMN) bit. This bit, if turned on, indicates that the accessed variables are not shared and thus the snoops need not be sent to 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 ESP technique. This can be compared against Figure 7 SSP technique where counting Bloom filters take the role of compiler in reducing the snoop probes for the variables other than stack. As shown in Figure 9, when the SMN bit is set, the load that reaches the last level cache does not snoop the lower level caches of the 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*). Similarly, the compiler annotates the store addresses to indicate if the program contained self-modifying code. In the case of self-modifying code, the SMC snoop probes (line denoted by *esp-smc* to the instruction cache is sent. The lines *esp* and *esp-smc* indicates the essential snoop probes. Compilers can use a multitude of techniques to generate the SMN bit. Using data flow analysis and algorithms [7], inter-procedure optimization, and other techniques, compilers can determine whether variables in a program are shared by other threads/programs. For example, in the OMP [13] construct used in multi-threaded programming supported by many compilers [12, 27], variables are explicitly declared as shared, private, etc. To support parallel programming, POSIX thread interface provides a better way of dealing with thread local storage (TLS) [8] using the *__thread* C/C++ keyword. This new keyword allows thread specific variables to be easily distinguished from the others. In addition to the compiler techniques, programming languages also provide scope for the variables. Java has global scope for classes, package scope for fields, methods within package, and procedure scope for local variables, and nested block scope. Similarly, the C/C++ language also provides the storage scope. This scope information can also be used by the compilers to determine if a variable is shared. Also, each thread has its own private stack to work with that is not visible outside. Using a combination of the techniques described above, compilers can determine if a variable in the program is shared or not. This gives the possibility to not snoop the other cores on a read or write to any variable that reach the last level cache to minimize the number of snoop probes. To achieve this, first the compiler performs data flow analysis and determines the semantics of the variables used in the program. It can also take inputs in the form of pragmas from the programmer if the program used any self-modifying code. 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 legacy code or code generated by a compiler that does not support SMN bits.

7. EXPERIMENTAL RESULTS

To evaluate the proposed techniques, we modified an x86 platform simulation infrastructure provided by Intel to model our SSP and ESP techniques. The detailed cycle accurate simulator models a hypothetical future CMP system. The simulator executes Long Instruction Traces (LIT) [14, 39] collected from the real-world applications including external events such as DMA and interrupts. The LITs are gathered for various categories: SPEC INT 2006, SPEC FP 2006, server, games and multimedia, and multi-threaded applications. In the multi-core configuration multiple copies of the same application are executed on each core, except for the multi-threaded categories. The example applications in each category are shown in Table 2. Each

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

category has 10 applications, and each application has multiple traces that represent different characteristic portion of the application similar to SimPoint [32] methodology. The simulator is executed for 100 Million instructions that cover many characteristic portions of the application after warming up all the caches, TLBs, predictors, and other hardware structures. The simulated configurations are 2-core, 4-core, 8-core, and 2x4-core (2 processor, 4-cores per processor).

A total of 500 traces are run and the simulated processor configuration is shown in Table 3. We used CACTI 4.2 [5] 70nm model to determine the energy consumed by the instruction cache, data cache and the hash arrays to evaluate the energy savings.

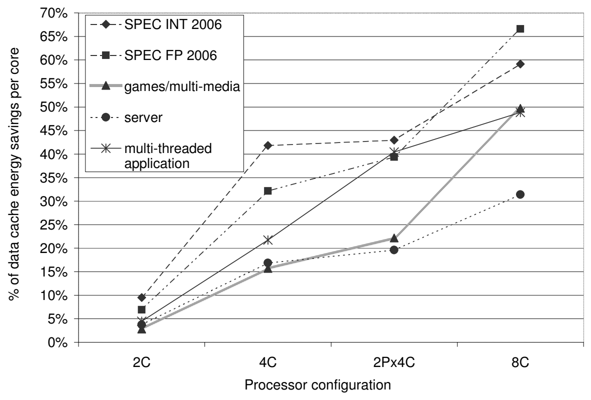


Figure 10: Energy savings in data cache per core using SSP

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 2C configuration and 30% to 65% in 8C configuration is achieved. It also shows that the data cache energy savings increase as the number of cores on the die increases. The reason is that as the number of cores increase, the number of snoops to all the cores also increases proportionately. The data cache energy savings in the 2Px4C dual-processor configuration is a little bit higher than the 4C (four core) configuration because of the external snoop (snoop probes between CMP processors) filtering in the 2P case.

Figure 11 shows the percentage of instruction cache energy savings using SSP technique. It shows that 50% to 70% of energy savings on average is achieved in the instruction cache across all processor configurations. The number of snoops to the instruction cache is determined by the number of writes and RFOs (Request-for-ownership) as noted in Table 1 and the number of store addresses that go through the LSB. The major contributing factor to the instruction cache snoops are the number of store addresses in the program. As the percentage of store addresses across different programs did not vary much, the percentage of energy savings in the instruction cache also do not vary widely.

Figure 12 shows the performance impact due to the SSP technique. It shows that on average there is nearly 1% to 2% performance improvement using SSP across various benchmark categories and different processor configurations. In one case, a 12% performance improvement was achieved as the reduced number of snoops to the lower level caches provided more oppurtunity to service the regular loads and stores that are needed by the core to make progress. Figure 13 shows the number of modified lines when the simulation completed. We also take into consideration the number of cycles required to flush these modified lines to the last level cache after the trace completion to support the eager writeback.

To evaluate the ESP technique we show the potential energy savings that can be achieved using the the hardware support that leverages the information generated by the compiler. The compilers can implement a variety of techniques to detect the sharing of the variables used in the

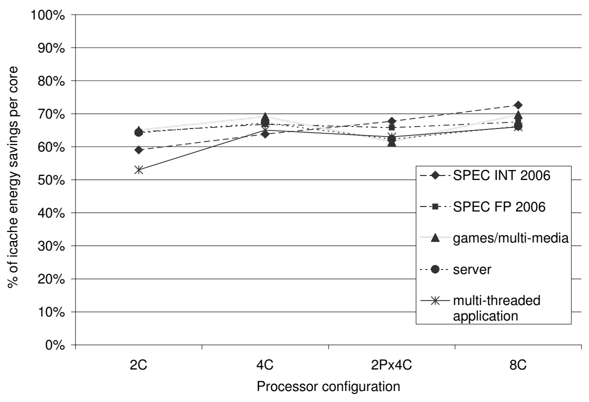


Figure 11: Energy savings in instruction cache per core using SSP

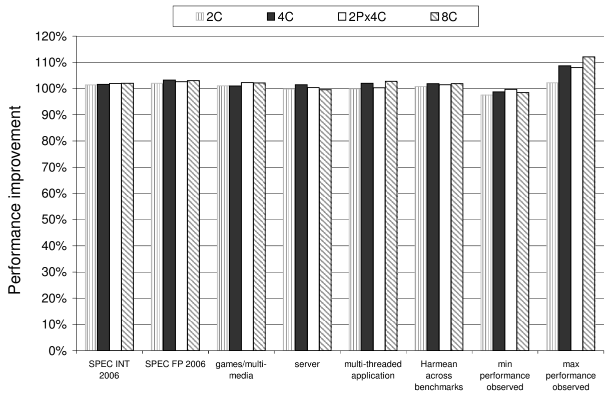


Figure 12: Performance impact using SSP

program. Figure 14 shows the percentage of cache energy that is spent on non-essential snoops. It shows that 5% (games category in dual-core configuration) to a maximum of 82% in the (SPEC FP 2006 in 8-core configuration) is spent on non-essential snoops that can be eliminated using the Essential Snoop Probe (ESP) technique. It also shows that as the number of cores in the CMP increases, the potential for energy savings also grows as the number of non-essential snoop probes are increased. Figure 14 shows that on average 85% of instruction cache energy is spent on the non-essential SMC snoop probes. This percentage of savings does not vary much across the benchmark categories and processor configurations. The ESP technique that uses the synergy between the processor and compiler can acheive higher energy savings.

8. RELATED WORK

It has been shown in earlier research work that many snoop requests miss in all the remote caches/nodes in a shared memory system [21, 37, 42, 24, 22]. But earlier work has not delved into the reason for these large misses in the other nodes. We found that one of the reasons is that the stack accesses do not have global visibility and snoop probes would miss in the other cores/processors. Another reason is that many programs do not share variables with the other programs/threads 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

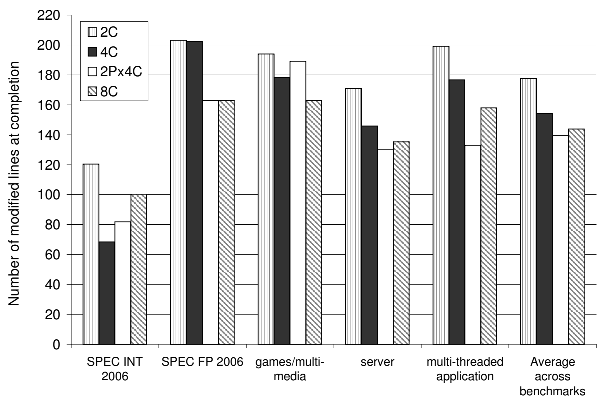


Figure 13: Number of modified lines after the program completion

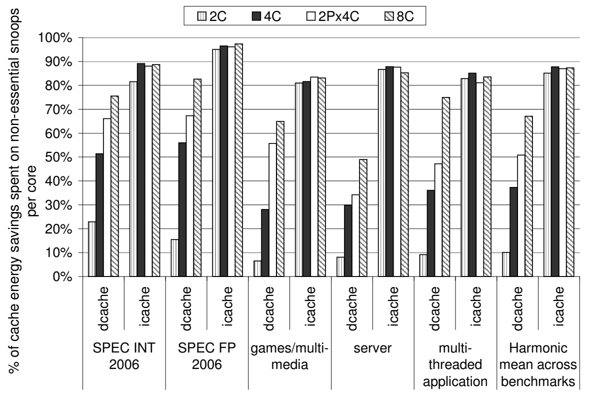


Figure 14: Energy savings in data cache and instruction cache per core using ESP

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 including pages and locality to optimize the snoops. In Page Sharing Table proposed by Ekaman [37] et al., used vectors to identify the sharing at page level. Saldanha et al., proposed serial snooping [42] to reduce the energy in shared multiprocessors. In Include Jetty [21] proposed by Moshovos et al., 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. According to the analysis done by Ekman et al in [29], they do not work as effectively when applied to CMPs: Include Jetty does not prevent a majority of the unnecessary snoops and Exclude Jetty requires prohibitively large hardware to work effectively. Our SSP and ESP techniques work well in a CMP enviornment as they selectively send snoop probes where needed.

Later Moshovos proposed RegionScout [20] where each node can determine in advance if a request would miss in all other nodes. In the RegionScout technique, whenever a node issues a memory request it also asks all other nodes whether they hold any block in the same region. If they do not, it records the region as not shared. Next time when the node requests a block in the same region, it now knows that it does not need to probe any other node. One key difference is that both the SSP and ESP techniques selectively send the snoop probes or completely eliminates if possible, whereas RegionScout has to broadcast the initial request to identify the Region Hit information. The RegionScout technique requires bus interconnect architectures as a wired-OR signal is necessary to notify a region hit. In contrast to RegionScout, our techniques are not limited by the choice of interconnect architecture used in the CMP system.

In [24], Cantin et al., 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. The SSP technique presented in this paper is different because we completely eliminate snoops for stack accesses and selectively dispatch snoops for non-stack accesses. In other words, whereas Cantin et al divided memory accesses into those requiring broadcast to all nodes and those that do not require broadcast at all, we generalize this division to include *exactly* which nodes to send the snoops to. Our ESP technique is completely different from RegionScout and Region Coherency Array as it uses compiler 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.

In [22] Atoofian and Banisadi propose a power-saving technique based on the obsevation that snoop responses exhibit high locality. They maintain saturating hardware counters to predict the likely node in an SMP system that will be providing a hit or hit-modified response to a snoop. Using this predictor they first send out a snoop to the predicted node and broadcast only if they get a clean response. Although Atoofian et al selectively send out snoops to the predicted nodes, unlike us they do not completely eliminate snoops for non-shared variables that constitue a major portion of snoops.

Snoop filters are being used in server platforms to separate each bus segment into different cache domains. IBM's X3 chipset uses a 48Mbits of eDRAM snoop filter that can cache the coherency information. Intel's Blackford chipset uses 16MB snoop filter to improve system performance. These integrated snoop filters use directory to track the state of all the cache lines in the processors to filter the coherency traffic and are not for filtering the internal snoops.

9. CONCLUSION

In this paper, we explored the snoops based on the semantics of the variables and variables that are shared in the program, as many different program categories can co-exist and run at the same time in the system. We exploited the fact that stack variables are not shared to relax the conservative cache coherence protocol and reduced the number of snoop probes. Using a hardware only technique called Selective Snoop Probe (SSP), snoop probes for the stack accesses were eliminated. In addition, counting Bloom filters were used based on MESI state transitions to reduce the number of snoop probes due to non-stack accesses. As a second technique, we proposed Essential Snoop Probe (ESP) to include all variables in the program by annotating the instructions with a Snoop-Me-Not (SMN) bit set by compilers representing the need to snoop during the execution.

One advantage of SSP technqiue is that all the functionality is implemented in hardware and is transparent to the programmer. Another advantage is that previously compiled binaries can benefit from this technique without the need for recompilation. As there is no information provided by the software, the energy saving achieved is limited in SSP. On the other hand, 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 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 energy is spent on the non-essential snoop probes that can potentially be saved using the compiler guided ESP technique.

In future work, we plan to extend the hardware and compiler based techniques to optimize systems like AMD Fusion, where both the regular and graphics programs run on their respective cores. In this kind of system, we expect the number of snoops initiated from the graphics cores to the other multiple cores on die to be much higher based on the nature of the graphics workloads. We expect our proposed techniques if used by compilers like CUDA (Compute Unified Device Architecture) can provide the required guidance to the hardware to optimize the snoop traffic, reduce the interconnect bandwidth and improve the overall power and performance of the heterogeneous system.

10. REFERENCES

[1] A developers guide to the POWER architecture. In *www.ibm.com/developerworks/linux/library/l-powarch.*

[2] Advanced Configuration and Power Interface. In *www.acpi.info.*

[3] AMD Athlon. In *www.amd.com/us-en/assets/content-type/white-papers-and-tech-docs/24659.PDF.*

[4] AMD K5 Technical reference manual . In *www.amd.com/es-es/Processors/TechnicalResources.*

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

[6] CMP Implementation in Systems Based on the Core Duo. In *download.intel.com/technology/itj/2006/volume10issue02/vol10-art02.pdf.*

[7] Codesurfer. In *www.grammatech.com/products/codesurfer/overview.html.*

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

[9] Intel 64 and IA-32 Architectures Software Developer's Manual. In *www.intel.com/design/processor/manuals/253668.pdf.*

[10] Intel Architecture Optimization Reference Manual. In *http://developer.intel.com/design/PentiumII/manuals/245127.htm.*

[11] Intel Itanium Architecture. In *www.intel.com/design/itanium/manuals/iiasdmanual.htm.*

[12] Intel OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance. In *http://developer.intel.com/technology/itj/2002/volume06issue01/vol6iss1-hyper-threading-technology.pdf.*

[13] Multiprocessor validation of the Pentium Pro microprocessor. In *www.openmp.org/drupal/mp-documents/spec25.pdf.*

[14] Performance analysis and validation of the Intel Pentium 4 processor on 90nm technology. In *www.intel.com/technology/itj/archive/2004.htm.*

[15] 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.*

[16] 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.*

[17] Power5 Architecture. In *www-941.ibm.com/collaboration/wiki/display/LinuxP/POWER5+Architecture.*

[18] Unleashing the Cell Broadband Engine Processor. In *www-128.ibm.com/developerworks/power/library/pa-fpfeib/, Nov. 2005.*

[19] Mrinmoy Ghosh and Emre Ozer and Stuart Biles and Hsien-Hsin S. Lee. Efficient System-on-Chip Energy Measurement with a Segmented Bloom Filter. In *ARCS '06*, 2006.

[20] Andreas Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. In *Proceedings of the 32nd International Symposium on Computer Architecture*, 2005.

[21] Andreas Moshovos and Gokhan Memik and Babak Falsafi and Alok N. Choudhary. JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers. In *HPCA '01*, 2001.

[22] E. Atoofian and A. Baniasadi. A Power-Aware Prediction-Based Cache Coherence Protocol for Chip Multiprocessors. *In the Third Workshop on High-Performance, Power-Aware Computing*, 2007.

[23] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communication ACM*, 13(7), 1970.

[24] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving multiprocessor performance with coarse-grain coherence tracking. In *Proceedings of the 32nd International Symposium on Computer Architecture*, 2005.

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

[26] Davib B. Gustavson. The Scalable Coherent Interface and Related Standards Projects. *IEEE Micro*, 12(1), 1992.

[27] Diego Novillo. OpenMP and automatic parallelization in GCC. In *GCC developers summit*, 2006.

[28] Dong Hyuk Woo and Mrinmoy Ghosh and Emre Ozer and Stuart Biles and Hsien-Hsin S. Lee. Reducing energy of virtual cache synonym lookup using bloom filters. In *CASES '06*, 2006.

[29] 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 '02.*

[30] Eric Dahlen and Jennifer Gustin and Susan Meredith and Doug Moran. The 82460GX Sever/Workstation Chip Set. *IEEE Micro*, 20(6):69–75, 2000.

[31] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-Fault Recovery for Chip Multiprocessors. In *Proceedings of the 30th International Symposium on Computer Architecture*, 2003.

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

[33] Hsien-Hsin S. Lee and Chinnakrishnan S. Ballapuram. Energy efficient D-TLB and data cache using semantic-aware multilateral partitioning. In *ISLPED '03*, 2003.

[34] Hsien-Hsin S. Lee and Mikhail Smelyanskiy and Gary S. Tyson and Chris J. Newburn. Stack Value File: Custom Microarchitecture for the Stack. In *HPCA '01*, 2001.

[35] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *ISCA '88*, 1988.

[36] Luiz Andre Barroso and Kourosh Gharachorloo and Robert McNamara and Andreas Nowatzyk and Shaz Qadeer and Barton Sano and Scott Smith and Robert Stets and Ben Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *ISCA '00*, 2000.

[37] Magnus Ekman and Per Stenstrom and Fredrik Dahlgren. TLB and snoop energy-reduction using virtual caches in low-power chip-multiprocessors. In *ISLPED '02*, 2002.

[38] Michael R. Marty and Mark D. Hill. Coherence Ordering for Ring-based Chip Multiprocessors. In *MICRO 39*, 2006.

[39] Peter G. Sassone and Jeff Rupley, II and Edward Brekelbaum and Gabriel H. Loh and Bryan Black. Matrix scheduler reloaded. In *ISCA-34*, 2007.

[40] Rakesh Kumar and Victor Zyuban and Dean M. Tullsen. Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling. *SIGARCH Comput. Archit. News*, 33(2), 2005.

[41] A. Roth. Store vulnerability window (svw): Re-execution filtering for enhanced load optimization. In *ISCA '05*, 2005.

[42] C. Saldanha and M. Lipasti. Power efficient cache coherence. *Workshop on Memory Performance Issuses in conjunction with ISCA '01.*

[43] Simha Sethumadhavan and Franziska Roesner and Joel S. Emer and Doug Burger and Stephen W. Keckler. Late-binding: enabling unordered load-store queues. In *ISCA '07*, 2007.

[44] M. S. Squillante and E. D. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2), 1993.

[45] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving Both Performance and Fault-Tolerance. In *Proceedings of the 9th International Symposium on*

Architectural Support for Programming Languages and Operating Systems, 2000.

[46] Weidong Shi and Hsien-Hsin S. Lee and Laura Falk and Mrinmoy Ghosh. An Integrated Framework for Dependable and Revivable Architectures Using Multicore Processors. In *ISCA-33*, 2006.