SEMANTICS-ORIENTED LOW POWER ARCHITECTURE

A Thesis
Presented to
The Academic Faculty

by

Chinnakrishnan S. Ballapuram

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
April 2008

SEMANTICS-ORIENTED LOW POWER ARCHITECTURE

Date Approved: March 28, 2008

To my loving wife,

beautiful daughter, and

wonderful parents.

ACKNOWLEDGEMENTS

I express sincere gratitude to my adviser, Dr. Hsien-Hsin Sean Lee for his support and guidance throughout the graduate career. He continuously motivated me to strive for the best and helped improve the quality of the dissertation. I learned the microarchitecture design from industry standpoint through his advanced RISC architecture classes that were lively with discussions. He also taught me to write good conference papers.

I want to thank all the MARS lab-mates for their friendship and thoughtful discussions we had. And many thanks to other Georgia Tech friends and class mates who made learning new things, working on projects, and other school activities enjoyable and easier. It might be strange, but I would like to thank few consolidated music streaming websites (saarega, shyamradio, olifm, raaga) that gave me patient and endless company on countless lone and calm nights for years.

I want to thank my wife, parents, and in-laws for their love and support. And many thanks to my daughter whose smile and presence gives joy.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Innovations in the microarchitecture and prominent advances in the semiconductor process technology enable designers to create more sophisticated and powerful microprocessors in all the market segments. At the same time, however, they also lead to increased power consumption. As more transistors become available on a processor die because of the process technology, two main design approaches are currently being taken to increase performance. One is to increase the levels of cache hierarchy and the sizes of the cache, and the other is to integrate more cores on the die to design chip multiprocessor (CMP). The capacity of caches in the microprocessors is increased to accommodate the new application workloads. The dynamic power in the lower level instruction and data caches (caches that are nearer to the core) and the leakage power in the last level caches dominates the power consumption in the memory sub-system. One of the problems facing CMP systems is the increasing number of snoops that increases with the number of cores on the die. The sophisticated superscalar techniques and snoop probes to the cache hierarchy exacerbate the power consumption and the thermal budget of a processor. Among all components of the processor, the instruction and data cache accesses, and the address translations consume a major portion of the overall power in a processor. When the future generation processors increase the levels of cache hierarchy, size of the caches, and the number of cores on the die, the power consumption problem becomes worse. This dissertation deals with the microarchitecutre techniques to alleviate the power consumption in the memory sub-system.

The main contribution of the thesis is the demonstration of Semantics-Oriented Low Power Architecture techniques that use the semantics of memory references and variables used in an application program to reduce the power consumption in the memory sub-system of a microprocessor. The first four techniques are called Semantic-Aware Mutltilateral Partitioning (SAM), Synonymous Address Compaction (SAC), Entropy-based Speculative and Deterministic TLBs, and Java instruction TLB (J-iTLB) and Java data TLB (J-dTLB). These four techniques target the Translation Lookaside Buffers and caches. The fifth technique has two parts: 1) Selective Snoop Probe (SSP), and 2) Essential Snoop Probe (ESP). Both of them aims to reduce the power consumed by the snoop probes that access the instruction and data caches in the CMP.

The *Semantic-Aware Multilateral Partitioning (SAM)* technique reduces the power consumption in the data TLBs and the data caches. The power savings is achieved by decoupling the data TLB lookups and the data cache accesses, based on the semantic regions defined by the programming languages and the application binary interface, into discrete reference sub-streams, namely, stack, global static, and heap. A new hardware-based mechanism called *Synonymous Address Compaction (SAC)* exploits the high locality of concurrent and consecutive memory operations that often access the same memory page to reduce the number of TLB lookups. It reduces power with minimal impact to the hardware budget. A novel *Entropy-based SPeculative (ESP-TLB)* mechanism for the stack references and *Entropy-based DeTerministic (EDT-TLB)* mechanism for the global static references was proposed based on the following two observations. The entropy of stack virtual page numbers is low resulting from good spatial locality of memory addresses that go to the stack. The entropy of higher-order bits of global virtual page numbers is low, as the size of the global data is determined and fixed during the program compilation.

The *J-iTLB* and *J-dTLB* structures exploit the memory reference characteristics and the interactions between the JVM and the Java applications. The method horizontally partitions the traditional monolithic instruction TLB and data TLB into distinct, smaller TLBs for special purposes targetted for Java applications. The J-iTLB scheme improves power and performance by eliminating the conflict misses between the JVM code and the Java application accesses in the instruction TLB. The J-dTLB organization splits the data TLB into two, for reads and writes, based on the observation that the optimized/unoptimized code is written into the heap space using the data TLB.

The final contribution deals with the snoop probes in the chip multiprocessor systems. We analyze the internal and external snoop behavior in a CMP system based on the semantics and sharing of the program variables. Based on the observations and the analyses, 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 program variables. By selectively sending the snoop probes, the above two techniques relax the conservative nature of the cache coherency protocol and its implementation to reduce power and improve performance.

CHAPTER I

INTRODUCTION

The modern microprocessors employ sophisticated and powerful microarchitecture techniques to improve performance. However, the techniques also lead to increased power consumption. As more transistors become available on a processor die because of the process technology, two main design approaches are currently being taken to increase performance. One is to increase the levels of cache hierarchy and the sizes of the cache, and the other is to integrate more cores on the die to design chip multiprocessor (CMP). The capacity of caches in the microprocessors is increased to accommodate the new application workloads. The dynamic power in the lower level instruction and data caches (caches that are nearer to the core) and the leakage power in the last level caches dominate the power consumption in the memory sub-system. The increase in the number of snoop probes to both the instruction and data caches in a CMP system to maintain the cache coherency further increases the power consumption. The caches, because of their large capacity, can consume around 40% [83] of the overall processor power. Current microprocessors devote a large fraction of the die area to memory structures. For instance, 30% of Alpha 21264's die area and 60% [81] of StrongARM's die area are devoted to cache and memory systems. As the combined size of the first level (instruction and data caches) and the middle level caches in CMP systems is now comparable to the size of the last level cache, reducing power consumption in all the levels of cache hierarchy is important in CMP systems. For example, the latest quad-core Barcelona [30] processor has 64KB of instruction cache, 64KB of data cache, and 512MB of middle level cache per core. This gives a combined capacity of 2.5MB of first level and middle level caches compared to a 2MB last level cache.

The superscalar processors decode, allocate, issue, and execute multiple instructions per cycle. They rely on accurate branch predictors, multiple address generation units, and multi-ported TLB structures to avoid stalls because of resource conflicts and to keep the processor supplied with instructions and data. Most of the processors including embedded systems support virtual memory, where Translation Lookaside Buffer (TLB) is an inevitable component in the processor design. The TLB is a content addressable memory that expedites the virtual-to-physical address translation for each virtual address accessed. The TLB draws a considerable amount of power, as it is typically designed as a fully- or highly-associative structure. It is accessed upon every instruction and data fetch. Measured data [67, 69] from commercial processors such as Intel's StrongARM and Hitachi's SH-3 reported that as much as 17% of the total on-chip power is consumed by the TLBs and the trend is increasing. Also, Ekaman [51] reported the TLB power consumption to be 20-25% of the total cache power consumption, and Fan *et al.* [53] reported 13% for a unified TLB.

In addition to the instruction execution, processors are required to maintain data consistency and memory ordering in the application program, where cache coherency protocols are used for this purpose. The cache coherency protocol sends snoop probes to the instruction caches, data caches, and other related internal buffers in the processor to work on the latest copy of the data. Most of the current microarchitecture studies use CMP as the baseline to analyze problems and to improve performance. The CMP systems can also be used to address emerging issues such as security [99], reliability [57, 104], 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 the design complexity. One of the problems facing CMP systems is the increasing number of snoops that increases with the number of cores on the die. Another factor contributing to the increased power consumption in the instruction and data caches, and in the bus/ring core interconnects, is the need to support and service snoop probes in multiprocessor systems. Unfortunately in CMP systems, the number of snoop probes to the caches increases with the number of cores per die and the levels of cache hierarchy.

Among all components of the processor, the caches and address translations consume a major portion of the overall power in a processor. As the superscalar techniques, like out-of-order execution, are now being used in the embedded domain [59], power and thermal control is a first-class priority in designing both the next generation of embedded and high performance microprocessors. The sophisticated superscalar techniques and snoop probes to the cache hierarchy exacerbate the power consumption and the thermal budget of a processor.

The thesis focuses on architectural techniques to reducing the power consumed by the memory sub-system resulting from the regular program memory accesses and the snoop probes from other cores/processors in a CMP/MP system. The mechanisms and techniques exploit the semantics of memory references and the characteristics of program variables to reduce power with minimal performance impact.

1.1 Thesis contributions

The five main contributions to this thesis are the following:

- We addressed the high energy consumption in the data TLB using a streamlined memory architecture partitioning technique called *Semantic-Aware Multilateral Partitioning (SAM).* It reduced the energy consumption in the memory sub-system without compromising performance. The energy savings is achieved by decoupling the data TLB lookups and the data cache accesses, based on the semantic regions defined by the programming languages and software convention, into discrete reference sub-streams, namely, stack, global static, and heap. Their unique access behaviors and the locality characteristics are analyzed and exploited for energy reduction.

- We analyzed the access pattern of memory operations performed within a cycle and in successive cycles, and exploited the characteristics of the addresses for energy reduction opportunities. In particular, we found that concurrent and consecutive memory operations demonstrated high locality and are often *synonymous*, accessing the same memory page. We proposed a new complexity-effective hardware-based mechanism called *Synonymous Address Compaction (SAC)* that exploited synonymous behavior to reduce the number of TLB lookups and reduce power with minimal impact to the hardware budget. The mechanisms can also be used to reduce the number of TLB ports, where the die area is limited.

- Based on the analysis of the TLB accesses, we made two observations with respect to the information content of the virtual page numbers (VPNs). First, the entropy or information content of the stack VPN is low because of good spatial locality of memory addresses that go to the stack. Second, the entropy of the higher-order bits of global VPNs is low, as the size of the global data is determined and fixed during the program compilation. Based on the two observations, we proposed a novel *Entropy-based SPeculative - Translation Lookaside Buffer (ESP-TLB)* mechanism for stack references and *Entropy-based DeTerministic - Translation Lookaside Buffer (EDT-TLB)* mechanism for global static references to reduce the overall data TLB power with minimal impact to the performance and hardware overhead.

- We analyzed the interactions and conflicts between the JVM and the Java application memory references. The *J-iTLB* and *J-dTLB* structures exploit the memory reference characteristics and horizontally partitions the traditional monolithic TLB into distinct, smaller TLBs for special purposes targetted for the Java applications. The J-iTLB scheme improves power and performance by eliminating the conflict misses between the JVM code and the Java application accesses in the instruction TLB. The J-dTLB organization splits the data TLB into two, for reads and writes, based on the observation that the optimized/unoptimized code is written into the heap space using the data TLB.

- The final contribution regards the snoop probes in CMP systems. The current trend in the industry is to increase the number of cores per die in every new generation of process technology. One of the problems facing CMP systems is the growing number of snoops necessary to maintain cache coherency and support self/cross-modifying code. These are power consuming and performance limiting events. To address these issues, we analyzed the internal and external snoop behavior in a CMP system based on the semantics and sharing of the program variables. Based on the observations and analyses, 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 the program variables. The techniques relax the conservative nature of the cache coherency protocol and its implementation by selectively sending snoop probes to reduce power and improve performance.

1.2 Thesis organization

The thesis is organized as follows:

- Chapter 2 starts with a brief overview of the virtual memory, the TLBs, and the snoop probes in a CMP system.
- Chapter 3 discusses the memory reference behavior of application programs. It is followed by the *Semantic-Aware Multilateral (SAM)* partitioning technique that exploits the memory accesses to different regions in the virtual address space to reduce the energy consumption in the TLBs and caches.
- Chapter 4 analyzes the access pattern of memory operations performed within a cycle and in successive cycles, and quantifies the frequency of synonymous accesses

in the data TLBs and the lookup redundancy. Two energy-efficient microarchitecture mechanisms, *intra-cycle compaction* and *inter-cycle compaction* are proposed to eliminate redundant lookups in the data TLB. These two techniques are based on the *Synonymous Address Compaction (SAC)* of memory references.

- In Chapter 5, the entropy content of the VPNs for the data TLB lookups is quantified. Based on that, we propose a novel *Entropy-based SPeculative TLB - (ESP-TLB)* mechanism for stack references and *Entropy-based Deterministic TLB - (EDT-TLB)* mechanism for global static references. These two techniques use the low information content in the VPNs to reduce the energy consumed by the TLBs without compromising the performance.
- In Chapter 6, we study the memory reference characteristics of a Java application running on a JVM and its interactions between them and propose an energy efficient instruction TLB and data TLB mechanisms.
- In Chapter 7, we address the increased number of snoop probes in the CMP-SMP (CMP based SMP) system by proposing two novel techniques called *Selective Snoop Probe* and *Essential Snoop Probes*. These two techniques relax the conservative cache coherency protocol and its implementation to reduce power and improve performance by selectively sending snoop probes.
- Chapter 8 discusses prior related research work and analyzes the differences between our proposed techniques and the earlier techniques.
- Chapter 9 concludes the thesis work and discusses other opportunities, in which our proposed techniques can be used.

CHAPTER II

MEMORY SUBSYSTEM AND CACHE COHERENCE

2.1 Virtual memory and TLB

The footprint of software applications is increasing in size along with the hardware enhancements. Virtual memory support is provided in most recent microprocessor systems, including the embedded systems to run bigger applications and to concurrently run many applications. The virtual memory separates the user memory from the main memory. The separation is achieved by dividing the main memory into equal-sized blocks called pages or unequal-sized blocks called segments. These pages or segments are allocated to different processes in the system. Each page in the system is uniquely identified by two page numbers, the virtual page number (VPN) and physical page number (PPN). There is a one-to-one correspondence between the virtual page number and the physical page number. This unique mapping is provided by a mapping table called a page table. The page table maps the bigger user address space to the smaller main memory space. In a virtual memory system, the CPU produces virtual addressses that are uniquely translated to physical addresses to access the main memory. The one-to-one mapping of a virtual address to a physical address is called memory mapping or address translation.

Figure 1 shows the address translation mechanism using the page table for a 32-bit virtual address. The virtual page number is the higher-order bits of the virtual address without the page offset. The virtual page number indexes the page table and retrieves the physical page number along with other attributes of the page. The uniquely determined physical page number is concatenated with the page offset and is sent to the main memory for memory access. The example shown in Figure 1 uses a

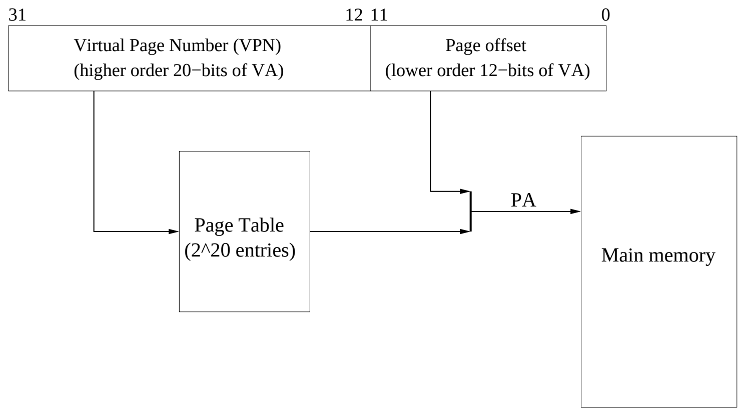

31
12 11
0
Virtual Page Number (VPN)
(higher order 20−bits of VA)
Page offset
(lower order 12−bits of VA)
Page Table
(2^20 entries)
PA
Main memory

Figure 1: Virtual to physical address translation using page table.

32-bit virtual address, a 4KB page size, and four bytes per page table entry. This gives a 4MB $((2^{32}/2^{12})\text{x}(2^{2}))$ page table. The page tables are stored in the main memory and are paged themselves. Paging means that every memory access now takes twice as long, one translation to obtain the physical address and the other to get the data, increasing the latency of the memory reference. The Translation Lookaside Buffer is a hardware structure that stores the most frequently used address translations to avoid the first latency. The TLB stores address mapping information and provides virtual-to-physical translation for each virtual address being accessed. On a TLB miss, the lookup is routed to the main memory and the corresponding address mapping is then reloaded into the TLB. The misses in the TLB result in page faults, which invoke the OS. The OS uses its own page table or other mapping structure to find a valid translation or create a new one and updates the TLB using the supervisor-mode instructions that can replace and update the entries in the TLB. This method of handling a TLB miss is called the software TLB miss handler. On the other hand, a hardware TLB miss handler uses a hardware state machine for accessing the memory to search the page table and provide translations for the misses. The processor system architecture specifies the page table organization structure. As the latency cost is expensive to memory, the TLB is normally designed as a fully- or highly-associative content addressable memory (CAM). However, due to this organization and the fact that the TLBs are accessed upon each instruction and data fetch, they can draw a considerable amount of power. There are normally two TLBs in a processor, one dedicated to the instruction and the other to the data. These two buffers are called instruction TLB (iTLB) and data TLB (dTLB). Modern processor architecture support small and large page sizes. The typical page sizes are 4KB for small pages and 4MB for large pages. The support for large pages reduce the pressure on the TLBs and sometimes on the data caches too. As the page table size becomes smaller with large pages, it reduces the conflict between the page tables and the application data in the data cache.

2.2 Snoop probes in chip multiprocessor

Cache coherence protocols must be maintained to guarantee data consistency for correctness in a multiprocessor system. Large-scale multiprocessor systems based on distributed shared memory usually employ a directory-based coherence scheme for data consistency. While, small- or medium-scale multiprocessor systems commonly use a shared bus or a ring architecture, where an invalidation-based or update-based snoop coherence protocol is implemented to make the caches coherent. All transactions go through a commonly shared bus or ring interconnect. The difference between the invalidation-based and update-based policies lies in whether the shared cache lines should be invalidated or updated when a processor writes to the same memory block. In the invalidation-based protocol, the shared cache lines held by the other processors are all invalidated, whereas in the update-based protocol, data are updated in all the caches sharing the same memory block. Variants of the MESI [88] protocol are implemented in commercial microprocessors, including Intel and AMD. The internal snoops to the lower level caches are inevitable, as the last level cache is shared by several cores on the same die in a CMP or multicore system. The snoop probes are not only necessary to maintain cache coherency, but are also required to support self-modified code (SMC) and cross-modified code (XMC). In the next section, snoops resulting from the SMC/XMC, snoop flows, snoop probes, and triggers in a typical CMP are described.

2.2.1 Snoops resulting from 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 [1, 4, 13, 14] provide support for self/cross-modifying code. There are many applications that use this feature. One example is Just-In-Time compilers that use SMC to generate optimized code amid runtime. To provide this type of support, on every write to a memory location in a code segment that is currently cached or prefetched, the processor must invalidate the associated cache line in the instruction cache and in the prefetch buffers. For example, in the Pentium 4 processor, if a write to 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 detect such a behavior, each store address needs to send a snoop probe to the instruction cache. If a match is found, the corresponding cache line in the instruction cache is invalidated. 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 a CMP are also sent to support cross-modified code (XMC).

2.2.2 Snoop flows

Figure 2 shows the flow of internal and external snoops in a typical quad-core CMP. Each core has its own private L1 instruction (iL1) and data (dL1) caches, and all four cores share one shared last level L2 cache (sL2). The shared L2 is accessed by the L1 instruction or data cache misses from all four cores, the L1 and L2 prefetchers, and external snoops in an MP system. The CMP interconnect that connects all the

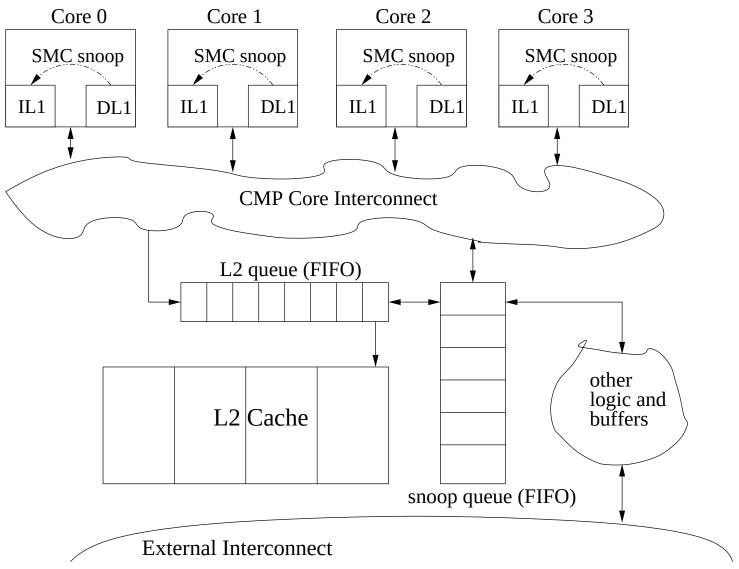


Figure 2: Snoop flows in a quad-core CMP system.

lower level cores can be a bus, a ring, or an arbiter-based interconnect. The requests that need to access the L2 cache are queued in a common hardware structure called the *L2 queue*. The L2 access is typically pipelined. The internal and external snoop requests are queued in another hardware structure called the *snoop queue* [7, 48]. 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, and MSHRs (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 of all cores. The snoop response, and data if needed, are propagated to the requesting cores. It is also necessary to perform 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 for performance considerations.

Snoop flows differ based on the type of CMP interconnect architecture used. In a

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

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	snoop
Code Fetch	Event Trigger	SMC snoop	snoop store buffer only	snoop	-	XMC snoop	snoop store buffer only	snoop	SMC snoop
Shared L2 evict	-	snoop	-	snoop	-	snoop	-	snoop	snoop
On store address dispatch	SMC snoop to iL1	-	-	-	-	-	-	-	-

ring interconnect, the snoops are sent across the ring, and all the 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 the snoop probes and return the responses to the requesting core as quickly as possible or to avoid the snoop probes completely whenever possible to improve the overall system performance.

2.2.3 Snoop triggers and snoop probes

Table 1 shows the internal snoop trigger points and the hardware structures that need to be snooped to support the 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 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 RFOs (request-for-ownership) 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 that reaches the last level cache is much more expensive than a data read or RFO (request-for-ownership) as it sends snoops to all the modules shown in the snoop table, but is mitigated by the lower instruction cache miss rates. Table 1 clearly shows that as the number of cores per die and the vertical cache hierarchy increases, the internal snoops will likely become bottlenecks in performance and power consumption.

This concludes the overview of the virtual memory, the TLBs, and the snoops in a CMP system. The following chapters will describe the microarchitecture techniques used to reduce the power consumption in the caches and the TLBs based on the semantics of memory references and program behavior in a multiprocessor system.

CHAPTER III

SEMANTIC-AWARE MULTILATERAL PARTITIONING

3.1 Motivation

The footprint of software applications is increasing in size along with the hardware enhancements. Virtual memory support is provided in most recent microprocessor systems, including the embedded systems to run larger applications and to concurrently run many applications. According to the convention of the programming languages and the application binary interface, each processor architecture typically partitions a virtual address space into several non-overlapped semantic regions. The three main semantic regions are instruction region, static data region, and dynamic data region. The static data is composed of global and read-only data allocated at compile-time. The dynamic data is composed of stack and heap regions. The stack region holds the activation record created at runtime for local variables, passing arguments, returned values, etc., during subroutine calls. The heap region is used by the dynamically allocated storage created by functions such as malloc() in the C programming language. As we will show, the semantic data regions created for different purposes demonstrate different characteristics that can be exploited with dedicated hardware.

The data memory reference patterns are analyzed to investigate the properties. Figure 3 and Figure 4 illustrates the footprint distribution of the data memory accesses. It is based on the semantic regions defined by the software convention using cjpeg from the MiBench benchmark suite as an example. Each point in the figure represents a data reference hit to a particular memory address. The lower-order 12 bits of data address are plotted on the x-axis and the higher-order 20 bits on the y-axis. In other words, the y-axis shows a *virtual frame number (VFN)* for a page size of 4KB (2^{12}). All the points drawn on a particular horizontal line belong to the same virtual memory page. The left figure plots the stack accesses and the right figure plots both the global static and heap accesses.

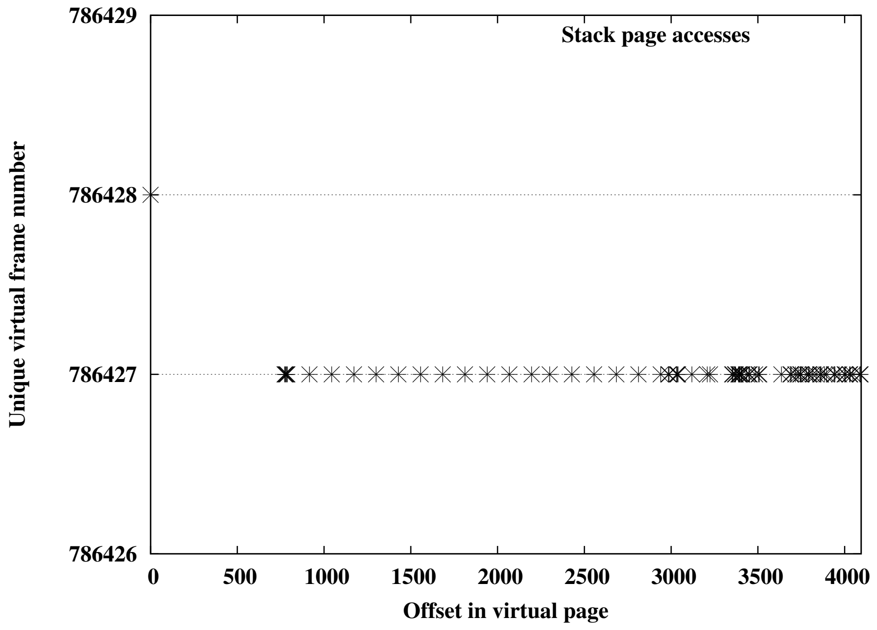


Figure 3: Dynamic address footprint distribution of stack region.

The figures show that the addresses within a particular semantic region form a *semantic band*. It clearly shows that the *heap band* is wider and denser than the *stack* and *global static bands*, as the number of unique heap pages accessed is substantially higher than those in the stack and global static regions. Also, note that only a small number of stack virtual pages is needed relative to the number of global static and heap pages.

Based on the observation in Figure 3 and Figure 4, we now examine the number of compulsory misses in the d-TLB and in the data cache for programs from the MiBench and the SPECint2000 benchmark. The programs from the MiBench were run to the end, while four selected programs from the SPECint2000 suite were run to three billion instructions. The MiBench benchmark suite is composed of representative embedded applications collected from networking, security, telecommunication, image

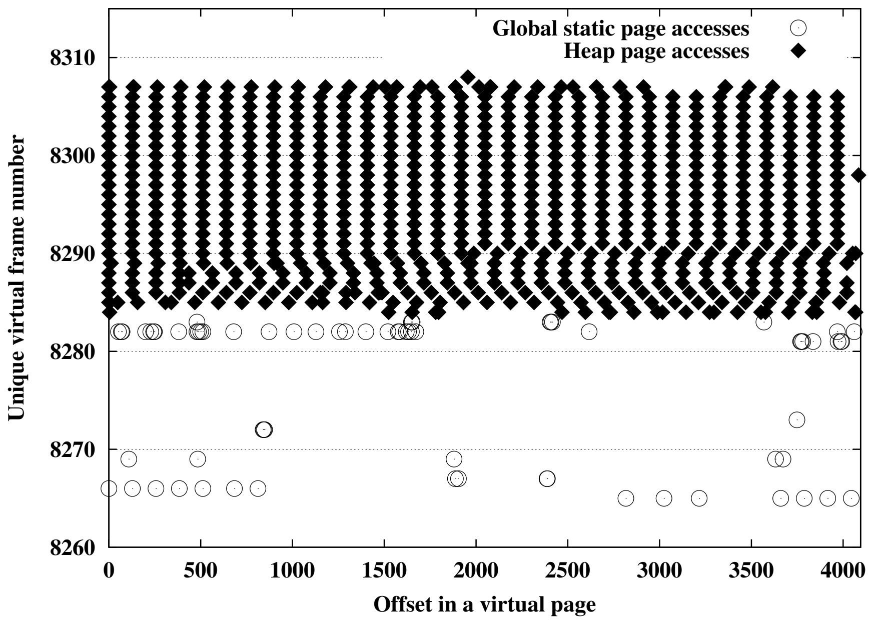


Figure 4: Dynamic address footprint distribution of heap and global regions.

processing, etc. The SPEC2000 integer benchmark suite is often used for evaluating high-performance systems. All the programs were compiled into an ARM binary with an `-O3` compiler switch.

Figure 5 shows the number of compulsory d-TLB misses, the number of unique virtual data memory pages accessed by a program in each semantic region. The memory page size used is 4KB, which is a typical page size used in modern operating systems. Note that the data is plotted in log scale because of the large number of heap references. Both benchmark suites show a similar trend of stack memory page utilization. The stack accesses can be satisfied by fewer than five pages for most of the programs. The number of global static pages is dependent on the behavior of a given application. Nonetheless, it is also significantly less than the number of heap pages, averaging 15 pages.

Figure 6 shows the number of compulsory data cache misses, where trends similar to the d-TLB are observed. It is worthwhile to note that the SPECint2000 benchmark programs demonstrate huge compulsory misses in the heap region compared to those

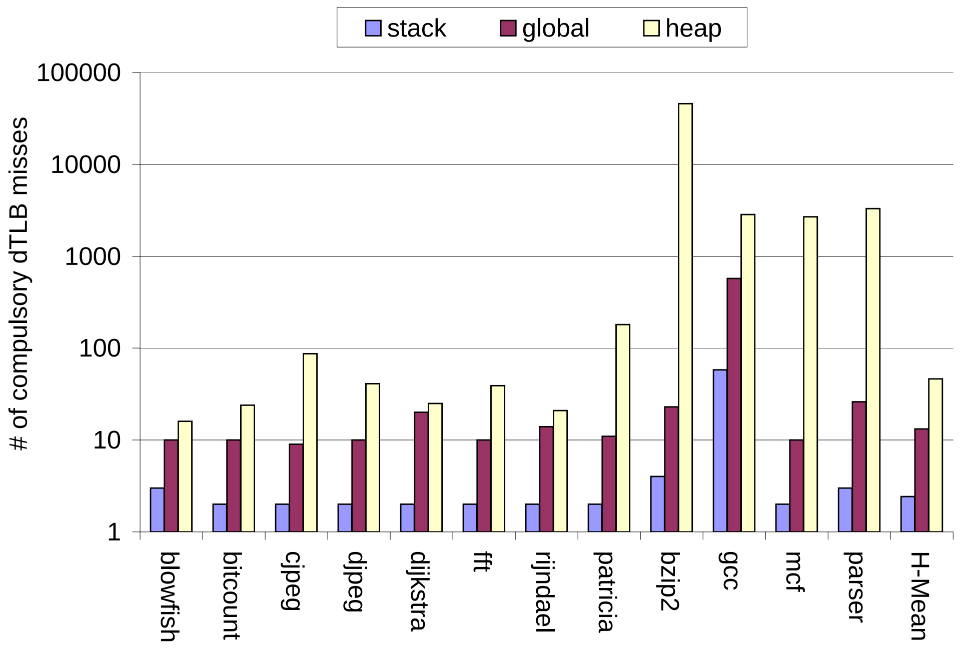


Figure 5: Compulsory data TLB misses.

shown for the MiBench programs. This suggests that the data working sets tend to expand at a faster rate in the heap region than in the stack and the global regions. We now quantify the distribution of dynamic data references in Figure 7.

In Figure 7, the number of accesses to each semantic region is normalized to the total number of data references. The critical information obtained from Figure 7 is that the majority ($\sim$50%) of dynamic memory accesses go to the stack. Figure 3, Figure 4, Figure 5, and Figure 6 together indicate that 1) data references largely hit the stack region, and 2) these stack accesses concentrate on very few memory pages. These properties lead to a new opportunity in reorganizing the data memory architecture for energy and performance optimization.

3.2 Semantic-Aware Multilateral Partitioning

We propose a new partitioning scheme for memory architectures called Semantic-Aware Multilateral (SAM) partitioning. The idea is to leverage the data reference characteristics and disperse the data memory accesses into discrete SAM sub-streams. Each stream is redirected to its own exclusive architectural component. The SAM

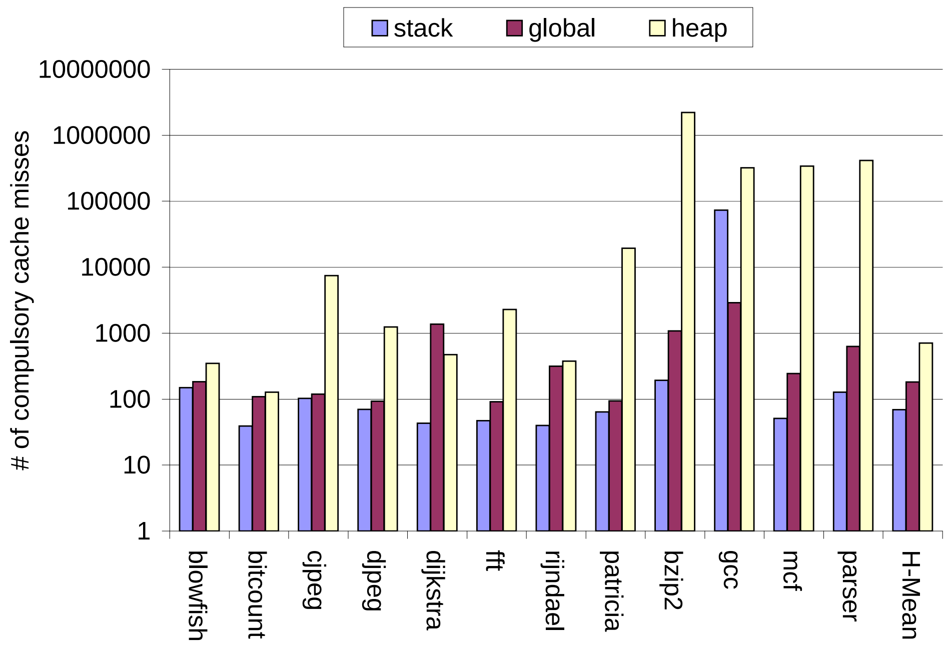


Figure 6: Compulsory data cache misses.

memory architecture, as shown in Figure 8, exploits the locality and characteristics demonstrated in each semantic region by 1) reorganizing the first level TLB structure into two small structures, namely, stack and global static micro-TLBs [43], leaving the second level [45] for all the data addresses, and 2) splitting the first level cache into three cachelets. The SAM TLBs and the SAM cachelets are hereafter referred to as SAT and SAC, respectively.

3.2.1 Semantic-Aware Data TLBs

In the SAM partitioning approach, we implement a Data Address Router (DAR) that routes each d-TLB lookup to a stack TLB (sTLB), a global static TLB (gTLB), or a heap TLB (hTLB) for address translation based on the higher-order bits of the virtual address. Note that the hTLB is also used as the second level TLB for the sTLB and gTLB. When loading a program for execution, the system loader communicates the following virtual addresses: the *ld_environ_base* (top of stack), the *ld_data_base* (global static base), and the *ld_data_bound* (heap base) to special hardware control registers. The DAR logic routes each address to its destination based on the ranges

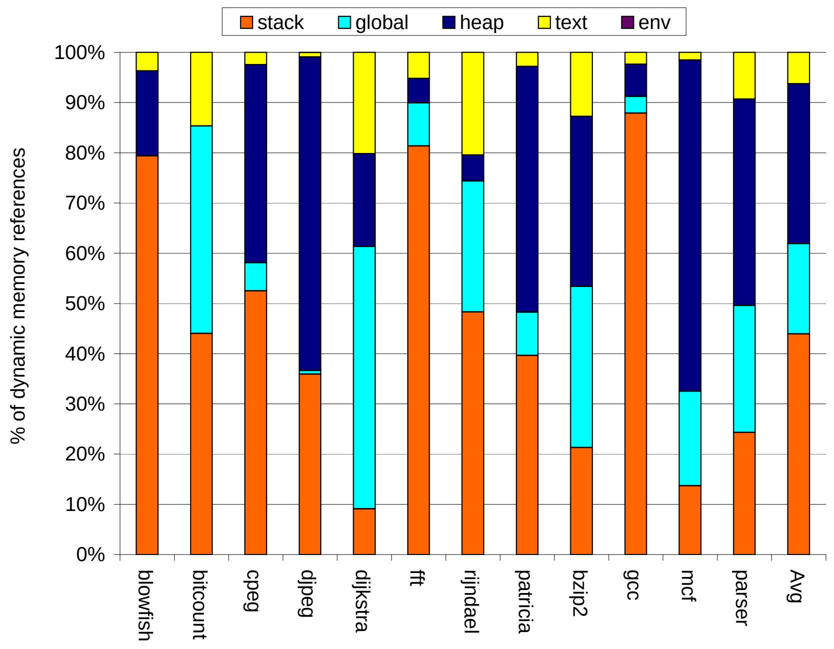


Figure 7: Dynamic data memory distribution for Mibench and SPECint2000.

derived from the control registers. Instead of searching the entire fully-associative TLB entries, energy can be saved by filtering out the lookups based on the stack and the global static regions. The lookups missing the two level TLB are directed to the main memory. If the virtual address does not fall under the stack or global static region, it is directed to the heap TLB. For example, the upper portion of virtual memory used by the kernel does not fall in the stack or global static region and is directed to the heap TLB.

Figure 9 shows the number of TLB misses and the miss rates when discrete SATs are implemented. The 181.mcf program of the SPECint2000 benchmark is used as an example for analysis. The number of misses is plotted on the log scale. The number of TLB entries on the x-axis is exponentially increased from one to 512 to measure the sensitivity of the TLB miss behavior. Not surprisingly, the number of sTLB misses saturates when the number of entries reaches two, while the gTLB saturates at eight entries. On the other hand, the hTLB misses do not drop as drastically as its counterparts, containing over 53 million misses even with 512 entries. The observation

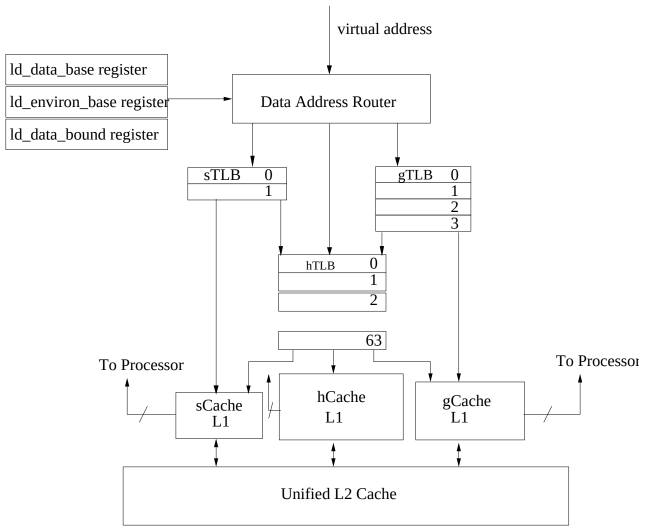


Figure 8: SAM memory architecture.

implies that under a unified TLB structure, the more dynamic heap TLB lines could evict the more stable stack and/or global TLB lines. The evictions lead to unnecessary TLB conflict misses that can be avoided by using a SAT implementation. Besides the advantage of eliminating the TLB conflict misses among different semantic regions, energy can also be reduced. The major portion of the memory access distribution is skewed toward the much smaller sTLB and the gTLB, leading to reduced energy savings. Moreover, multi-porting the micro-TLBs becomes more feasible when the die area and access latency are constraints. All the aforementioned advantages are quantified in the following sections.

3.2.2 Semantic-Aware Data Cachelets

The first level data cache can be semantically partitioned into discrete semantic-aware cachelets (SAC) similar to the semantic-aware TLBs (SATs). The energy savings could be substantial, as the energy dissipated in the caches constitutes a major portion of the overall energy consumption. Multi-porting the ever-increasing caches for superscalar processors also exacerbates leakage dissipation. The SAC scheme provides an alternative solution, selectively multi-porting only the highly accessed SAC,

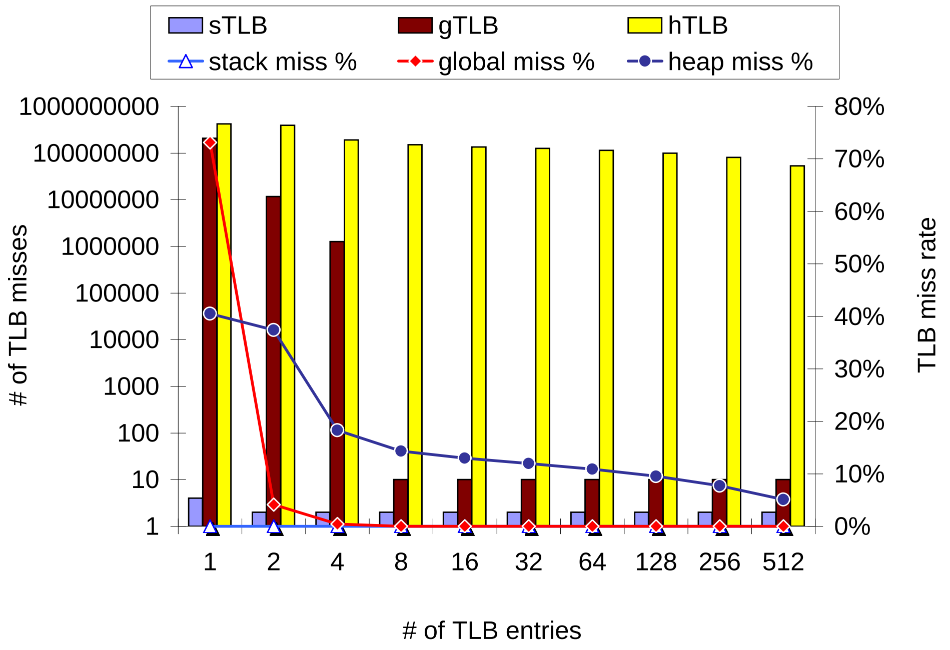


Figure 9: Semantic-aware TLB misses.

e.g., stack cache.

Figure 10 shows a graph similar to Figure 9 for the data caches. The size of each SAC is varied from 2KB to 256KB. Again, accesses to the stack region demonstrate a stable working set size with respect to the other two semantic regions. The global misses saturate at a reasonable capacity, whereas the heap data appear to be less tractable. Therefore, the majority of the memory references can be captured by semantically partitioning the cache structure into a small stack cache (sCachelet), a small global static cache (gCachelet), and a larger cache (hCachelet). The larger hCachelet serves the requests from the heap, text, and env regions. This new SAC scheme can substantially reduce the energy consumption while retaining the performance. The energy savings can be higher in multi-issue machines, as only the smaller sCachelet and gCachelet need to be multi-ported.

3.3 Experiments and Analysis

The simulation infrastructure is based on the Simplescalar simulator for the ARM ISA [27] for performance evaluation. We integrated the Wattch [38] power model with

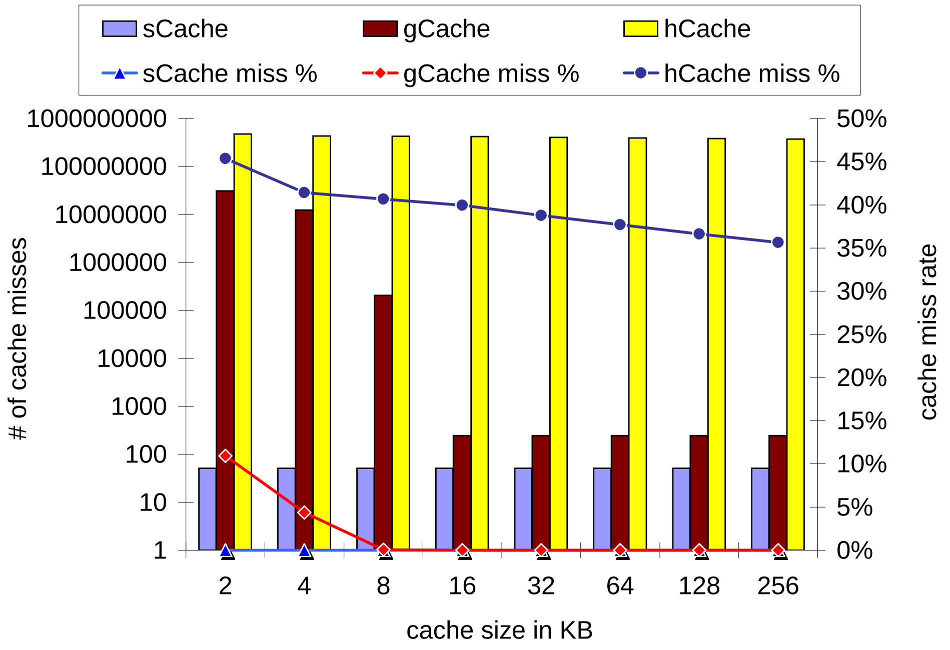


Figure 10: Semantic-aware cache misses.

the Simplescalar ARM model for energy simulation. We made the necessary changes to enable our studies for the SAM memory architecture. Table 2 lists the processor parameters used in our baseline machine model. We assume the leakage power consumption to be 10% of the dynamic power consumption for all the units. The SAC sizes used in the following sections are identical across all the benchmark programs. The SAT sizes are changed slightly for different benchmarks and are discussed in Section 3.3.1.

Table 2: Processor model parameters.

Execution Engine	out-of-order
Fetch/Decode Width	4 / 4
Issue/Commit Width	4 / 4
L1 cache hit latency	1 cycle
L2 cache hit latency	6 cycles
Memory latency	150 cycles
TLB hit latency	1 cycle
TLB miss latency	30 cycles
Cache property	Direct-mapped, 32B line
L1 Cache baseline	32KB
L1 s/g/hCachelet	8KB / 8KB / 16KB
L2 Cache	4-way 512KB, 32 bytes line

Table 3: Cost-effective TLB configuration.

Benchmark:	blowfish	bitcount	cjpeg	djpeg	dijkstra	fft	rijndael	patricia	bzip2	gcc	parser
dTLBbase	32	32	128	64	64	64	32	256	64	64	64
sTLB	2	2	2	2	2	2	4	2	4	4	4
gTLB	8	8	8	8	32	8	8	8	16	16	16
hTLB	16	32	128	64	32	64	32	256	64	64	64

Table 4: Multi-porting Configuration.

Number of d-TLB entries			
base	sTLB	gTLB	hTLB
64	4	16	64
2 ports	2 ports	1 port	1 port

Data cache sizes			
base	sCachelet	gCachelet	hCachelet
32KB	8KB	8KB	16KB
2 ports	2 ports	1 port	1 port

3.3.1 Performance vs. Energy with SAM

We ran a wide spectrum of simulations to collect the data for making the best selection of optimal TLB entries for each benchmark application. As discussed in Section 3.2, there exists a knee point, after which the performance gain diminishes regardless of the number of TLB entries added. Table 3 summarizes the most effective number of TLB entries for each benchmark. In most of the cases, the SAT approach has slightly more TLB entries than the baseline except for the blowfish, where the combined SAT size is slightly smaller than the baseline case. Figure 11 shows the performance speedup, the TLB energy savings, and the cache energy savings using the design parameters in Table 2 and the configuration in Table 3. The trends observed in Figure 11 for both the MiBench and the SPECint2000 are similar. An average of 36% energy reduction in the TLBs and 34% in the caches with 4% performance loss are achieved using the Semantic-Aware Multilateral partitioning technique.

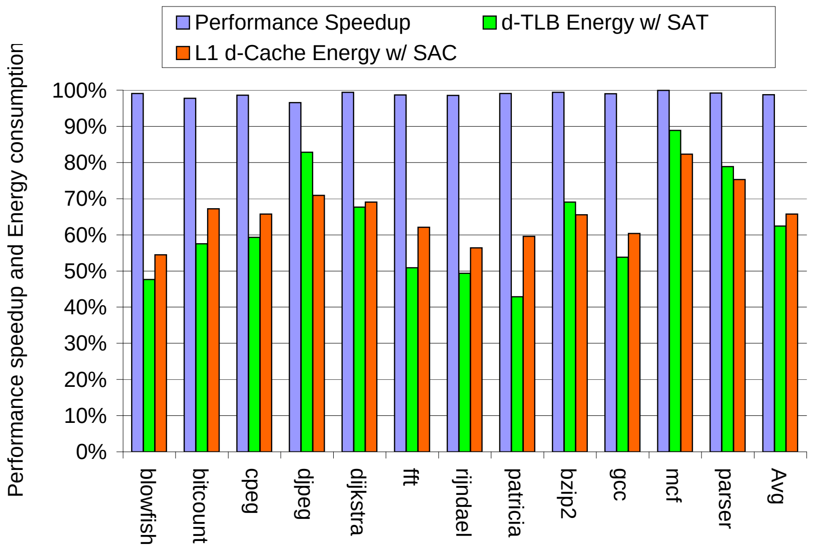


Figure 11: Design effectiveness of SAM (baseline=1.0).

3.3.2 Multi-Porting SAT and SAC

3.3.2.1 Performance vs. Energy

It was observed that one out of every two to three instructions is a memory operation in the contemporary applications and the workloads. This implies that a multi-issue machine needs to access memory (the dTLB and the data cache) more than once per cycle, necessitating a multi-ported design. The multi-porting of an ever-increasing monolithic TLB and cache aggravates the energy consumption and increases the die area. As shown in Figure 7, the average ratio of memory accesses to the stack, global, and heap is close to 2:1:1. In other words, we can multi-port the SAT and SAC using this ratio. Figure 12 shows the performance and energy savings for such a design using a four-wide SAM machine with the configuration listed in Table 4. It shows a 47% reduction in the data TLB and a 45% reduction in the data cache with 4% performance penalty.

3.3.2.2 Area and Access Latency Estimation

Table 5 shows the transistor areas for different SAC schemes using the CACTI 3.0 [66] tool. As shown in the table, selectively dual-porting the SAC reduces the access time

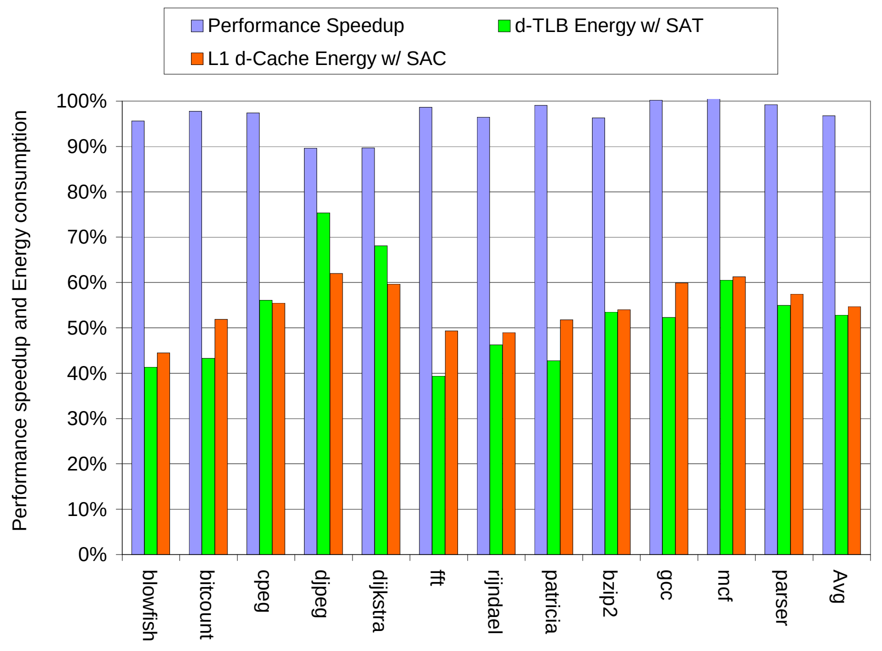


Figure 12: Multi-porting effectiveness of SAM (baseline=1.0).

by as much as 50% and transistor area by 32 to 41%. Also, note that the leakage energy (not shown in the table) can be saved proportionally as the transistor count is reduced.

3.4 Summary

We proposed a Semantic-Aware Multilateral (SAM) memory design technique that effectively reduced the energy consumption in the data TLB and the data cache with minimal performance impact. It is achieved by decoupling the data TLB lookups and the data cache accesses into discrete reference sub-streams, namely, stack, global static, and heap, based on the semantic regions defined by the programming languages and the software convention. Their unique access behaviors and locality characteristics are analyzed and exploited for energy reduction. As a result of the unique memory reference characteristics, the SAM approach reduced dynamic energy by redirecting the majority of accesses to the much smaller SA-TLB (SAT) and the SA-Cache (SAC). It was found that the number of SAT entries in the stack can be as few as two to cover almost all the stack references. We discussed the design of the data

Table 5: Access time and die area comparison.

Cache Model	32KB Unified	8KB sCachelet	8KB gCachelet	16KB hCachelet	Total SAC area	Area savings
R/W ports	2	2	1	1		
Access time (ns)	1.125	0.826	0.692	0.816		
Area in mm^2	5.304	1.393	0.616	1.095	3.104	41.5%
Model	64KB Unified	16KB sCachelet	16KB gCachelet	32KB hCachelet	Total SAC area	Area savings
Access time (ns)	1.630	0.949	0.816	0.948		
Area in mm^2	8.942	2.555	1.095	2.246	5.897	34.1%
Model	128KB Unified	32KB sCachelet	32KB gCachelet	64KB hCachelet	Total SAC area	Area savings
Access time (ns)	2.238	1.125	0.948	1.196		
Area in mm^2	17.24	5.304	2.246	4.115	11.666	32.3%

address router that splits and routes each access to the corresponding SAT and SAC. Using our technique, we showed that an average of 35% energy can be reduced in the d-TLB and in the data cache. The effect of multi-porting the SAT and the SAC were also investigated. The energy savings is 46% and the die area savings is 41% by multi-porting only the heavily accessed SAC and SAT instead of multi-porting their monolithic counterpart. The SAM technique is also orthogonal to prior vertical and horizontal partitioning schemes. Prior techniques such as filter caches or line buffers can be easily implemented on top of the SAM scheme to acquire more energy savings.

CHAPTER IV

SYNONYM ADDRESS COMPACTION

The multi-issue superscalar processors have become the de facto standard not only in the high performance computing platforms, but also in the embedded system platforms [59]. These sophisticated processors issue and execute multiple instructions per cycle. They rely on the accurate branch predictors, multiple address generation units (AGU), and multi-ported TLB and caches to keep the processor supplied with the instructions and data. Most caches are either physically indexed and physically tagged (PIPT) or virtually indexed and physically tagged (VIPT) based on the needs of virtual memory management and cache coherency maintenance. In both the cases, an address translation using the TLB is needed for each access. Multiple instructions issued in each cycle require multi-ported instruction TLB and data TLB to avoid stalls because of resource conflicts. Additionally, the TLBs are typically organized as a fully-associative cache to eliminate the latency intensive page walks resulting from conflict misses. As a result, the TLBs are often implemented as a CAM, where all the CAM cells are probed and compared to find a match on each TLB access.

We analyze the access pattern of memory operations performed within a cycle and in successive cycles, and exploit the characteristics of the addresses for energy reduction opportunities. In particular, we find that the concurrent and consecutive memory operations demonstrate high locality and are often *synonymous*, accessing the same memory page. As a result, a single TLB lookup is often sufficient to find the correct translation for multiple accesses in the same cycle, eliminating the redundant lookups that could draw additional power. Similarly, the most recently accessed data TLB entry can be latched and reused for the subsequent TLB lookups. We propose two new hardware-based mechanisms to reduce the number of TLB lookups. These mechanisms are complexity-effective and power-efficient with minimal impact to the hardware budget. In addition to reducing power, these mechanisms can also be used to reduce the number of TLB ports when chip area is a concern.

4.1 Motivation

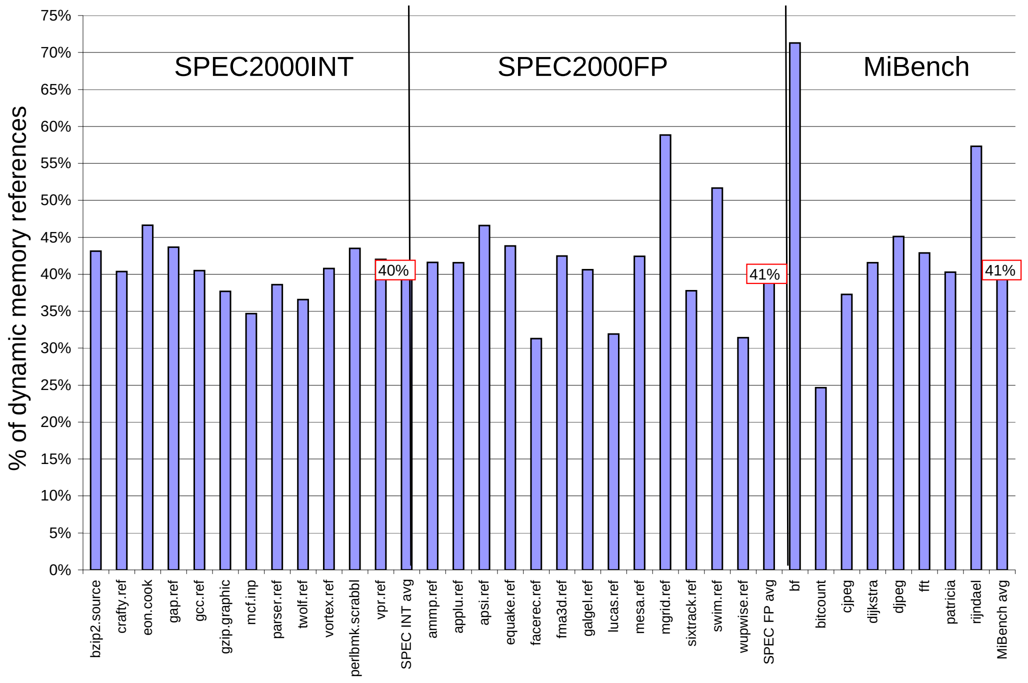


Figure 13: Memory references as a fraction of all dynamic instructions.

Using the MiBench [58] and SPEC CPU2000 benchmark suites, Figure 13 shows that more than 40% of the dynamic instructions executed in a program are memory references. We ran all the benchmark programs to completion for this experiment, except for the art that stopped at 500 billion instructions. The reference inputs were used for the SPEC2000. This study shows that, in a superscalar processor, multiple memory operations are performed concurrently in each cycle. To support these multiple memory instructions, multiple AGUs, larger memory order buffers, and multi-ported TLBs and caches are needed.

Now, we examine the distribution of dynamic memory accesses to study the behavior of memory references in a given cycle (intra-cycle) and in consecutive cycles (inter-cycle). A four-wide machine with 4KB memory page size is used in this experiment.

4.1.1 Intra-cycle behavior of memory references

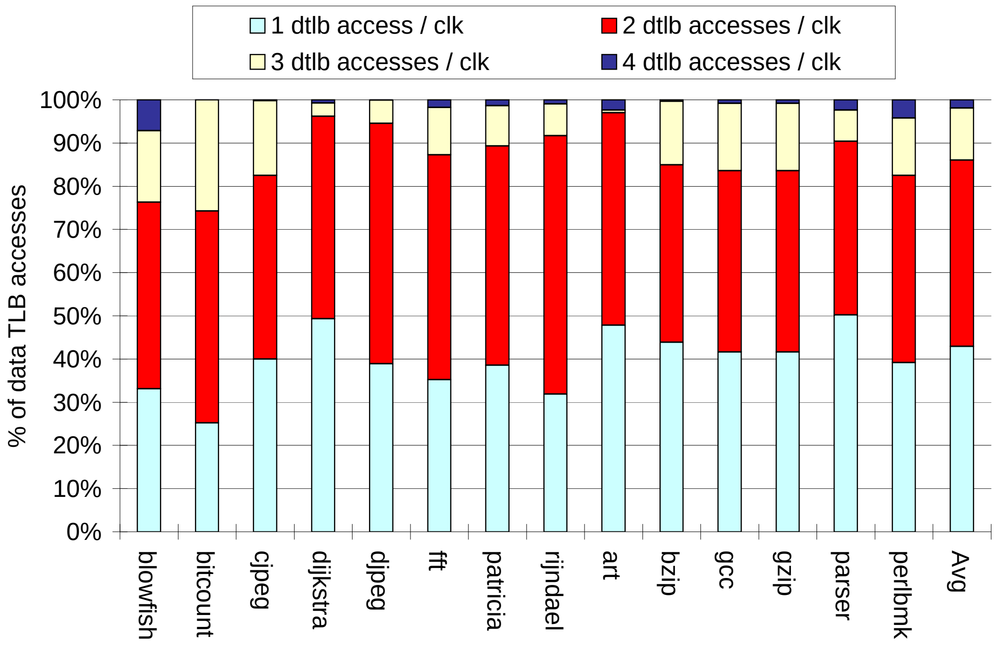


Figure 14: Breakdown of d-TLB accesses.

Figure 14 shows the breakdown of the data TLB accesses according to the number of concurrent references per cycle. On average, for 58% of the accesses, the processor issues more than one data TLB lookup in a cycle that requires a multi-ported TLB to avoid the stalls. An interesting property of these simultaneous memory accesses is that they often access the same page (using the same virtual-to-physical translation in the TLB). We call these accesses as *intra-cycle synonymous* accesses.

Figure 15 shows the breakdown of memory accesses according to the number of d-TLB access synonyms. In the figure, *syn(0)* means no access synonym, i.e., all memory references in the intra-cycle are unique. An access is *syn(1)*, when one another access in the intra-cycle is its synonym, i.e., the same page is used by two

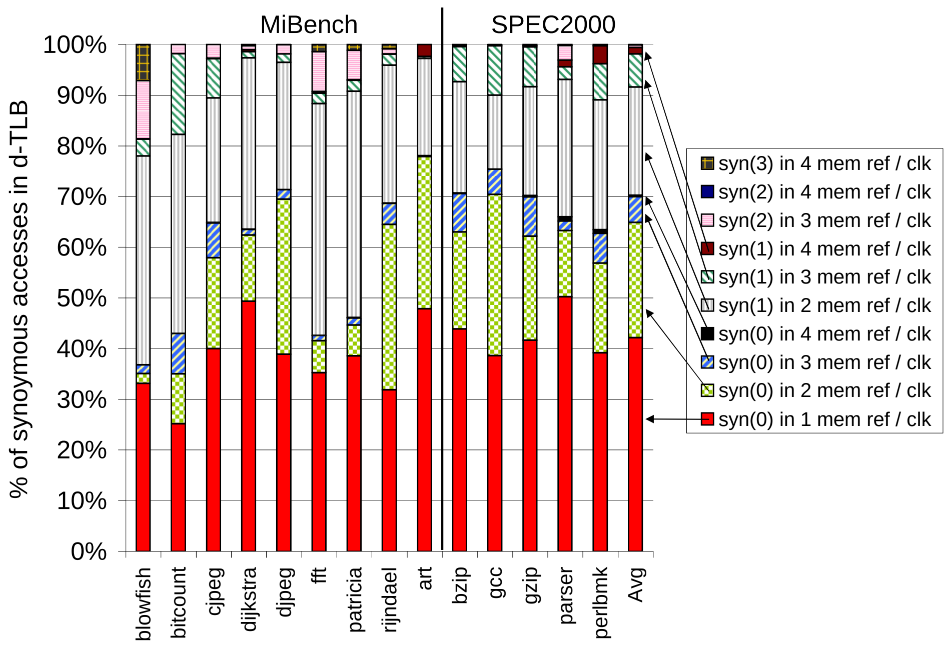


Figure 15: Breakdown of synonymous intra-cycle accesses in d-TLB.

memory references in the same cycle. More generally, *syn(N)* is when there are N other memory references in the intra-cycle accessing the same page. Within each *syn(N)* group, accesses are further broken down according to the number of memory references per cycle, yielding a total of ten categories. As the simulated machine is four-wide, a maximum of four memory accesses can be issued in each cycle. The bottom four segments in each bar represent the non-synonymous, i.e., *syn(0)*, accesses in one, two, three, or four simultaneous memory accesses per cycle. The next three bar segments represent the *syn(1)* accesses in two, three, or four memory references per cycle. Similarly, the next two bar segments show the portion of the *syn(2)* accesses, and finally the top bar segment represents the *syn(3)* accesses that can occur only when there are four memory operations issued in the same cycle.

As shown in Figure 15, 30% of the references have synonyms indicating that there is access redundancy that can be eliminated by using a single d-TLB lookup to satisfy these synonymous accesses. With this technique, we can remove the d-TLB lookups for $\frac{1}{2}$ of all the *syn(1) accesses*, $\frac{2}{3}$ of all the *syn(2) accesses*, and $\frac{3}{4}$ of all the *syn(3)*

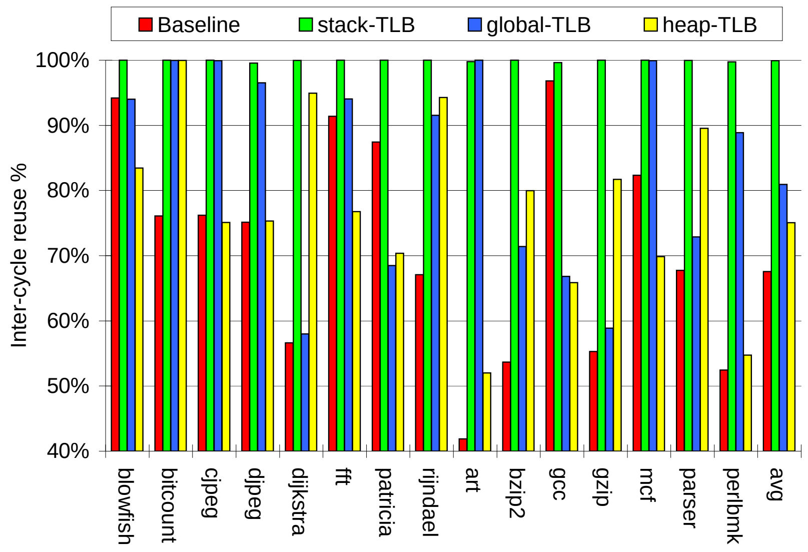


Figure 16: Inter-cycle reuse of d-TLB translations.

accesses.

4.1.2 Inter-cycle behavior of memory references

Another congruent memory reference behavior that occurs during program execution is when two consecutive memory references go to the same page. We call these accesses as *inter-cycle synonymous accesses.* The inter-cycle synonymous accesses can be exploited by a simple mechanism that detects the reuse of immediately preceding address translation. Our mechanism keeps the most recently accessed TLB translation and reuses it if the next access is synonymous. As shown in the leftmost bar (baseline) of Figure 16, 66% of accesses could reuse the last address translation using a fully-associative TLB. The reuse rate can be further increased if the data TLB is horizontally segregated into discrete differentially-sized TLBs based on the semantic regions, namely, stack, global, and heap, as proposed in the Semantic-Aware Multilateral Partition (SAM). As shown in Figure 16, the stack-TLB shows almost perfect inter-cycle reuse of nearly 99%, followed by 82% for the global-TLB, and 75% for the heap-TLB. In addition to improving the probability of reuse, these semantic-aware d-TLBs also allow reuse detection to be applied selectively, which will be shown in the experimental results section.

4.2 VPN Compaction Mechanisms

The intra-cycle and inter-cycle synonymous memory references provide an opportunity to reduce the energy consumed by the TLBs. In this section, we describe two orthogonal virtual address compaction techniques that make use of the synonymous properties. Figure 14 and Figure 15 show that 58% of memory references have companions in the same cycle and even three or four memory references per cycle are not uncommon, 14% on average. Furthermore, access locality can be high and synonymous accesses may look up the same memory page or even the same cache line in the same cycle or in the consecutive cycles.

4.2.1 Overview of VPN Compaction

Table 6: Virtual address access sequence.

cycle i	0xdeadbeee	0xdeadbeef	0xdeadbef0	0xffffffff
cycle (i+1)	0xdeadbef2	0xdeadbef3	0x12345678	—

Table 7: VPN translation lookup in d-TLB.

cycle i	0xdeadb	0xdeadb	0xdeadb	0xfffff
cycle (i+1)	0xdeadb	0xdeadb	0x12345	—

Table 6 shows an example of two back-to-back execution cycles for a four-issue machine with a 4KB page size and a four-ported d-TLB. The corresponding virtual page numbers looked up in the d-TLB are shown in Table 7. This example will be used in the following sections to illustrate our compaction mechanisms.

4.2.1.1 Intra-cycle compaction

The combining of multiple same-cycle synonymous lookups into one is called as *intra-cycle compaction*. This compaction is shown in Table 8, where the three lookups (for addresses 0xdeadbeee, 0xdeadbeef, and 0xdeadbef0 in Table 6) can be compacted into one in the first cycle (cycle i). Similarly, the two VPN lookups (for addresses 0xdeadbef2, 0xdeadbef3) can be compacted into one in the cycle (i+1). As shown, after intra-cycle compaction only two TLB accesses are needed in each cycle that saves power and reduces the number of required d-TLB ports. To perform intra-cycle compaction, a dedicated logic, to be discussed in Section 4.2.2.1, is designed to detect the access synonyms and eliminate the redundant d-TLB accesses.

Table 8: VPNs after intra-cycle compaction.

cycle i	0xdeadb	—	—	0xfffff
cycle (i+1)	0xdeadb	—	0x12345	—

4.2.1.2 Inter-cycle compaction

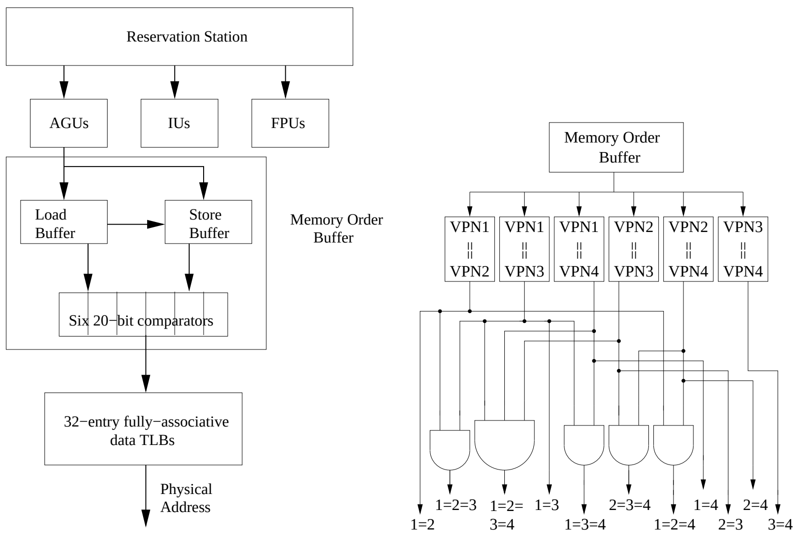
The reuse of address translations in consecutive cycles is called as *inter-cycle compaction*. Using the memory access example in Table 7, this technique latches the d-TLB translation used in cycle i for all the four addresses 0xdeadbeee, 0xdeadbeef, 0xdeadbef0, and 0xffffffff and reuses it for the addresses 0xdeadbef2 and 0xdeadbef3 in cycle (i+1). Table 9 shows address translations needed after inter-cycle compaction. Effectively, two d-TLB lookups are saved in cycle (i+1). We note that the reused translation remains latched and could be reused again as long as the same memory page is accessed consecutively. For example, Figure 16 shows that the stack address translation often fits into this category. The extra logic for inter-cycle compaction needs careful trade-off evaluation to ensure that the extra energy consumed does not exceed the energy saved by compaction.

Table 9: VPNs after inter-cycle compaction.

cycle i	0xdeadb	0xdeadb	0xdeadb	0xfffff
cycle (i+1)	—	—	0x12345	—

4.2.2 Implementation of VPN Compaction

4.2.2.1 Intra-Cycle Compaction



(a) Modified Microarchitecture (b) Comparator Logic

Figure 17: Architectural enhancements for intra-cycle compaction.

Based on the prior discussion, we design an intra-cycle compaction mechanism to eliminate the same-cycle synonymous d-TLB lookups. For an N-wide machine, we add C(N,2) comparators at the end of the memory order buffer, before addresses are used for actual memory accesses. For instance, a four-issue machine needs C(4,2)=6 comparators. The width of each comparator is same as the VPN width. The comparators are used in each cycle to eliminate the VPN redundancy and send only the unique VPNs to the fully-associative d-TLB for address translation. Figure 17(a) illustrates the proposed microarchitecture enhancement assuming a four-issue machine with a 4KB page size and a 32-bit address space. The comparator logic, detailed in Figure 17(b), asserts one of the output signals to indicate the degree of synonym for eliminating the redundant lookups.

In addition to saving energy, this technique also offers benefits in reducing the d-TLB lookup latency and the port requirement. For instance, in a four-issue machine, the d-TLB does not need to be designed for the worst-case, i.e. four-ported, as the occurrence of $syn(3)$ is rare for four memory references in a cycle, though $syn(2)$ is not uncommon, as shown in Figure 15. To avoid the performance loss, at least three ports are needed for the d-TLB. A d-TLB with two ports would be sufficient using our proposed intra-cycle compaction, as some of $syn(2)$ can be compacted to one or two memory references when the opportunity arises. A d-TLB with fewer ports is a smaller structure with a shorter lookup latency. This could be used to reduce the number of cycles needed for a d-TLB lookup when the operating frequency is high, or to increase the number of entries in the d-TLB without increasing the lookup latency.

4.2.2.2 Inter-cycle Compaction

The inter-cycle compaction mechanism can be implemented by simply latching the most recently used (MRU) TLB entry and its virtual page number. Later, the latch is read to detect reuse and obtain the translation. Figure 18 shows a semantic-aware memory (SAM) architecture [76] extended with this mechanism. The SAM architecture uses a Data Address Router (DAR) to decouple a single memory stream into the stack, global, and heap sub-streams. In this case, the semantic-aware TLB splits a conventional TLB into a two-entry stack-TLB, a four-entry global-TLB, and a 32-entry heap-TLB for exploiting the semantic-region affinity. In the same figure, inter-cycle compaction is enabled for each individual semantic-aware TLB. The number of reuse latches are same as the number of ports. Typically, TLBs are already designed with latches to test the read and write circuitry. During the address translation, the VPN of the virtual address is first compared against these latches. If the VPN matches (hit), the latched physical address is used. If the VPN does not match (miss), a lookup to the corresponding fully associative semantic-aware TLB will be performed. The performance impact of misses in the reuse detection circuitry is discussed in the next section.

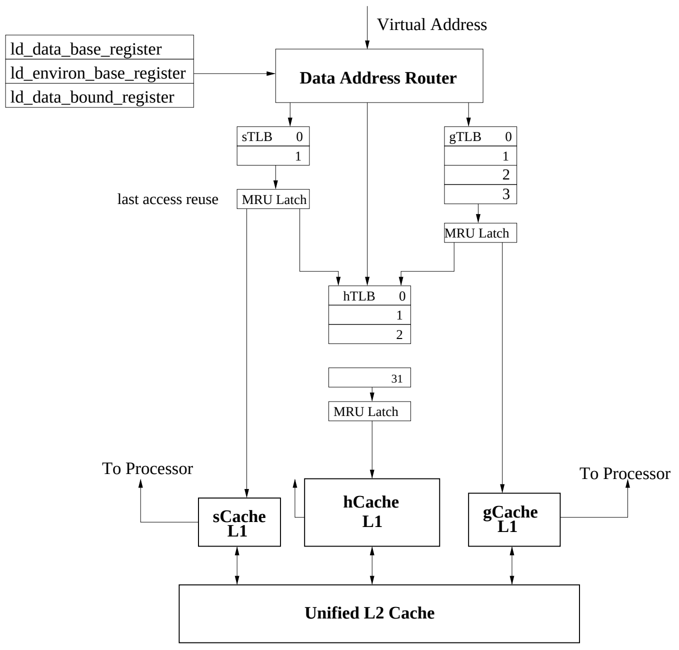


Figure 18: Semantic-aware memory architecture with inter-cycle compaction.

4.3 Experimental results

Our performance evaluation infrastructure is based on the Simplescalar simulator for the ARM ISA. We integrated the Wattch [38] power model with the Simplescalar ARM model for energy simulation. We made the necessary changes to enable our studies for the semantic-aware memory architecture and the compaction of synonymous virtual addresses. The MiBench simulations are run to the end. The SPEC2000 simulations are run for three billion instructions after fast-forwarding the first one billion instructions. The total memory references for these benchmarks vary from as low as 25% for the bitcount program to as high as 72% for the blowfish program in the MiBench benchmark and from 34% for the mcf program to 43% for the bzip2 in the SPEC benchmark. Table 10 describes our machine model.

To evaluate the energy savings using the intra-cycle compaction mechanism, the power consumption of the six 20-bit comparators used in eliminating the redundant TLB lookups are taken into account in the Wattch simulation. We designed the comparators in Verilog and synthesized them using the Synopsys Design Compiler

Table 10: Processor model parameters.

32-bit Processor Parameters	**Values**
Execution Engine	out-of-order
Fetch/Decode Width	4 / 4
Issue/Commit Width	4 / 4
Number of data TLB entries	32
Page size	4 KB
L1/L2 cache hit latency	1 / 6 cycle
Memory latency	150 cycles
TLB hit/miss latency	1 / 30 cycles
L1 Cache baseline	Directed-mapped, 32KB, 32B line
L2 Cache	4-way 512KB, 32B line
Number of TLB ports used	2
Each 20-bit comparator power	300μW
Each MRU latch power in TLB	140μW

targeted at 0.35μm technology. Each 20-bit comparator consumes 300μW and takes about 550ps based on our synthesized results. Similarly, each MRU latch comparison for the inter-cycle compaction mechanism consumes 140μW. The energy consumed by the comparators and the latches are added in each cycle, even if only one of the d-TLB access was issued. We charge one extra cycle for the six 20-bit comparators and one extra cycle for the MRU latch comparison. We also add an extra 10% of the dynamic power to account for the leakage power if there was no TLB access activity. The rest of the processor modules use the same 0.35μm technology scaling for power measurements using the Wattch.

4.3.1 Energy Reduction of Intra-cycle and Inter-cycle Compaction

Figure 19 presents the energy savings by applying the intra-cycle and the inter-cycle compaction mechanisms separately to the d-TLB. The baseline in the figure is a conventional 32-entry d-TLB. The rightmost bar shows the total energy savings attained by applying the intra-cycle compaction mechanism to the baseline d-TLB. On average, nearly 27% of d-TLB energy savings is achieved with 9% penalty, as shown by the last bar in Figure 20.

Figure 19 also shows the combinations of inter-cycle compaction mechanism applied to selective semantic-aware d-TLBs, as a trade-off for the miss penalty. The first

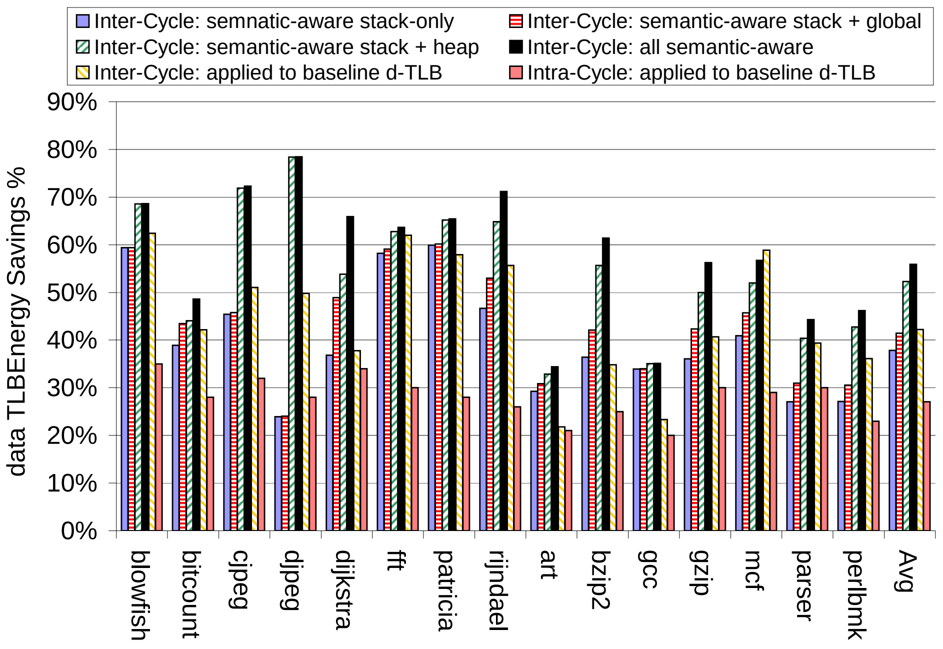


Figure 19: Energy savings using the intra-cycle and the inter-cycle compaction mechanism.

four bars show the energy savings achieved by employing stack-only, stack+global, stack+heap, and stack+global+heap using the inter-cycle compaction. In general, the energy savings reaches a maximum of 56% when inter-cycle compaction is applied to all the semantic-aware d-TLBs with less than 4% performance penalty (the 4th bar from the left in Figure 20). Using the inter-cycle compaction mechanism in a conventional d-TLB results in 42% energy reduction with 8% performance slowdown (fifth bar in Figure 19 and Figure 20). The inter-cycle compaction scheme can provide higher energy savings over the intra-cycle compaction when the reuse address translation hit rate is higher. Hence, most of the benchmarks show better performance using the inter-cycle compaction over the intra-cycle compaction scheme.

4.3.2 Synergistic Synonymous Address Compaction

The intra-cycle and inter-cycle compaction mechanisms can be combined together to further reduce the overall energy consumption. Hence, we apply the aforementioned two compaction mechanisms to the instruction TLB (i-TLB), as it also exhibits high

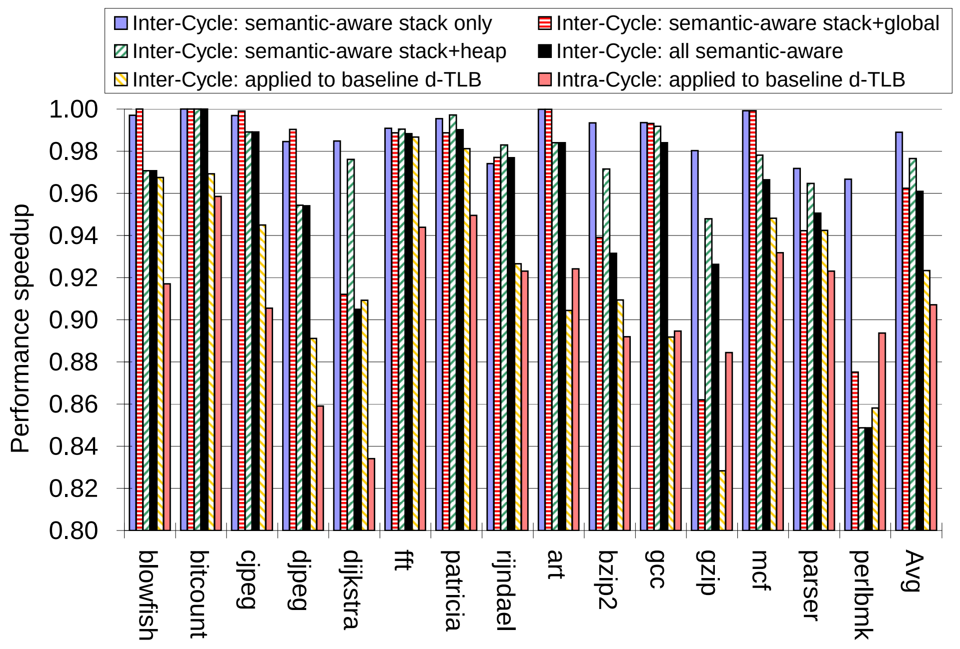


Figure 20: Performance impact of the intra-cycle and the inter-cycle compaction mechanism.

synonymous access behavior. Note that the instructions change memory pages only when a subsequent instruction crosses the page boundary or when a cross-page branch is taken. First, by applying both the compaction mechanisms to the i-TLB and the d-TLB separately, an average of 85% of the i-TLB energy and 52% of the conventional d-TLB energy is reduced, as shown by the first two bars in Figure 21. A more encouraging observation is that several benchmark programs even demonstrate more than 90% energy savings in the i-TLB. Next, we combine both the compaction schemes and apply them to both the i-TLB and d-TLB together to evaluate the overall TLB energy savings. As shown by the last two bars in Figure 21, 70% and 76% energy reduction is attained. It shows that applying both compaction schemes to the i-TLB and the semantic-aware d-TLB saves most of the energy spent in the TLBs.

4.3.3 Performance Impact for Combined Compaction

In the worst-case design, the two compaction schemes require two extra pipeline stages if the TLB lookup cycle budget cannot accommodate them. Figure 22 illustrates

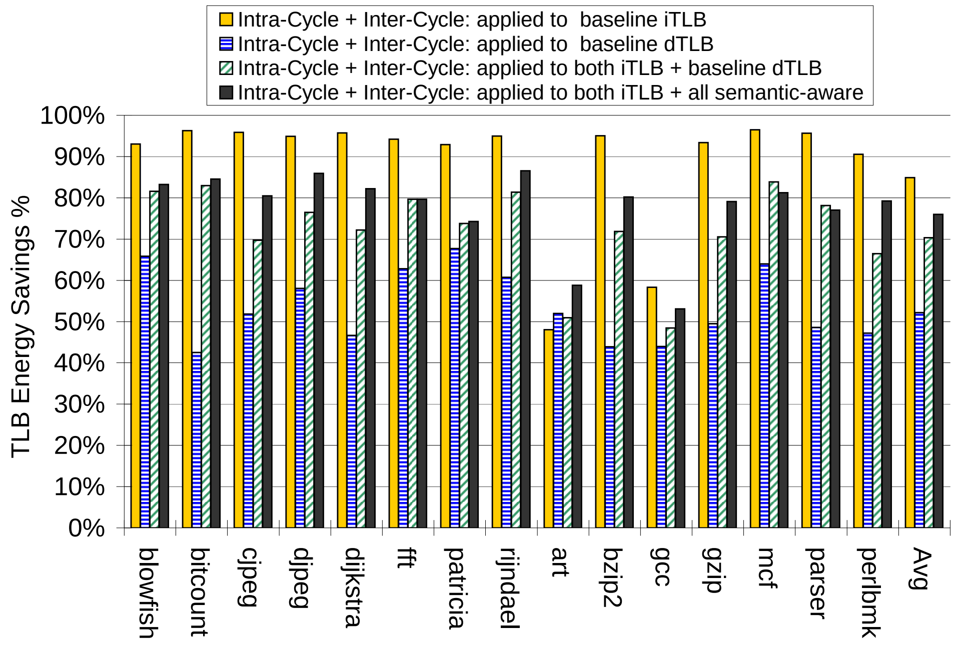


Figure 21: Overall i-TLB and d-TLB energy savings using both intra- and inter-cycle compaction mechanism.

the performance impact with the increased latency. In the worst case, 14% (3rd bar from left) performance was lost when the latency penalties for intra- and inter-cycle compactions are charged to both the i-TLB and the conventional d-TLB. The loss is reduced down to 11% (4th bar from left) when the conventional d-TLB is replaced with the semantic-aware d-TLBs, as the reuse address translations have a higher hit rate. If the compaction mechanisms are applied to only the i-TLB and d-TLB separately, the performance degradation is 5% and 13% (1st and 2nd bar), as shown in Figure 22. The performance overhead can be eliminated from the critical path for the i-TLB if the design can perform the comparison against the MRU latch as soon as the PC is updated before the subsequent fetch cycle.

4.4 Summary

We proposed the intra-cycle and inter-cycle synonymous address compaction techniques. It exploited the address translation redundancy to reduce the TLB energy.

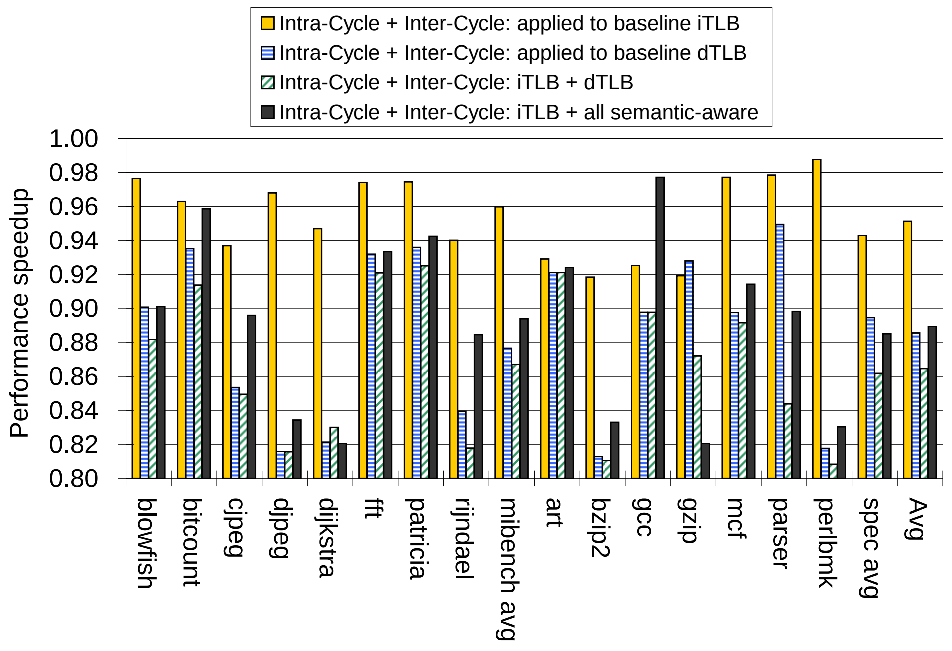


Figure 22: Overall i-TLB and d-TLB performance impact using the intra- and inter-cycle compaction mechanism.

It was found that the concurrent and consecutive memory accesses are often *synonymous*, going to the same memory page. The same-cycle synonymous accesses can be eliminated using an intra-cycle compaction mechanism that uses comparators to eliminate the redundant TLB lookups. Similarly, the inter-cycle synonymous accesses are eliminated by reusing the most recently used address translation stored in the latches. The address comparators and the MRU latches add little power overhead and access time, but allowing significant energy savings.

We also quantified the frequency of synonymous accesses in the d-TLBs. More than 30% of intra-cycle and nearly 76% of inter-cycle d-TLB accesses were synonyms. In the semantic-aware TLBs, the ratio is even higher: 99% for stack, 82% for global, and 80% for heap. To exploit the intra-cycle and inter-cycle synonyms, we proposed two energy-efficient microarchitecture mechanisms: 1) the intra-cycle compaction mechanism to eliminate the same-cycle synonymous accesses by using simple comparators, and 2) the inter-cycle compaction mechanism that reused the address translation with an MRU latch. The techniques resulted in an average energy savings of 27% and 42% using the intra- and inter-cycle compaction in the conventional d-TLBs with roughly 9% and 8% performance penalty. When the inter-cycle compaction technique was applied to the semantic-aware architecture, the energy savings was 56% with 4% performance penalty. We also evaluated our compaction techniques by combining the i-TLB and d-TLB together. The overall TLB energy savings was 70% for an i-TLB with a conventional d-TLB and 76% for an i-TLB with a semantic-aware d-TLB. In an area-constrained processor, our scheme can enable more memory operations per cycle without adding extra access ports in the TLB. Our simple hardware-only technique for saving energy is one way of exploiting the synonymous accesses.

CHAPTER V

ENTROPY-BASED DATA TLB DESIGN

The virtual page numbers exhibit a large degree of spatial and temporal locality. A given sequence of data accesses may map to the same physical page, because of the following two reasons: 1) spatial and temporal locality of the data accesses 2) the page size is typically much larger than the regular cache line. Therefore, little entropy or information is conveyed between consecutive virtual page number lookups in the TLB. Also, most of the applications do not utilize the entire address space. Therefore, the higher-order bits in the VPN remain unchanged, leading to less entropy. We analyze the entropy content of the VPNs exhibited in the memory reference stream and exploit their characteristics for energy reduction opportunities. Each memory access by a program carries a certain amount of information. The entropy or information content is a measure of the unexpectedness and varies with the program behavior. In particular, we find that the stack references have the lowest information content. Also, the stack references are highly predictable. As the number of the pages is fixed during the compile time for the global static data, most of the higher-order bits in the global data memory references do not change (zero entropy) during the program execution. Thus, a small number of VPN bits is sufficient for the address translation. We propose a novel *Entropy-based SPeculative - Translation Lookaside Buffer (ESP-TLB)* mechanism for the stack references and an *Entropy-based DeTerministic - Translation Lookaside Buffer (EDT-TLB)* mechanism for the global static references to reduce the overall data TLB energy. The mechanisms are complexity-effective and power-efficient.

5.1 Motivation

The entropy or the information content is a measure of “uncertainty” or “unpredictability” of a random variable X [98]. In other words, simple repetitive patterns contain low entropy. If X is a random variable that represents the VPN during the program execution, then the average amount of information or the zeroth-order Markov source entropy is given by

$H_0 = -\sum_{i=1}^{N} p(x_i)\ log_2\ p(x_i)$ ... (1),

where $p(x_i)$ is the probability of the occurrence of the VPN x_i, $-log_2\ p(x_i)$ is called self information, and H_0 is called average self information. The entropy has dimension in bits per VPN reference.

For example, in a virtual memory system, when the virtual address is 32-bits and the page size is 4KB (12-bits), the VPN size is 20-bits. The program can access any of the 2^{20} virtual pages during its dynamic execution.

1) In the first case, let us assume that there is only one memory reference for the entire program. Then the probability of accessing a virtual page is $1/2^{20}$. The self and average information is $-log_2\ (1/2^{20})$ bits/VPN reference.

2) Let us assume that all the virtual addresses go to the same page for all the references. The average information for this kind of VPN accesses is zero by applying the equation 1. Because there is only one x_i, the probability of occurrence of this x_1 is one, and log 1 is zero, we did not get any kind of useful information for this type of repetitive behavior.

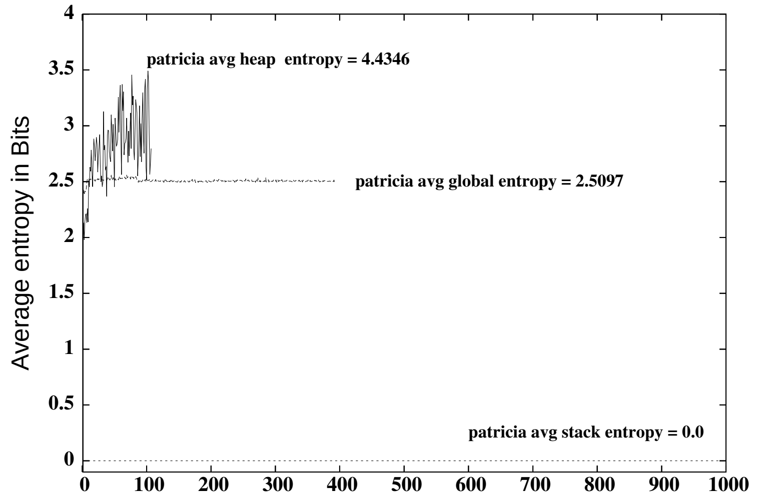
3) Now, in the third case, when each of the 2^{20} virtual address goes to a unique page, the average information is 20 bits/VPN reference by applying the equation 1. This is the maximum amount of information that can be obtained. Here, x_i varies from 0 to 2^{20}-1.

The first-order Markov source entropy is a measure of unpredictability of the next VPN given the knowledge of the previous VPN. The weighted average first-order entropy is given by

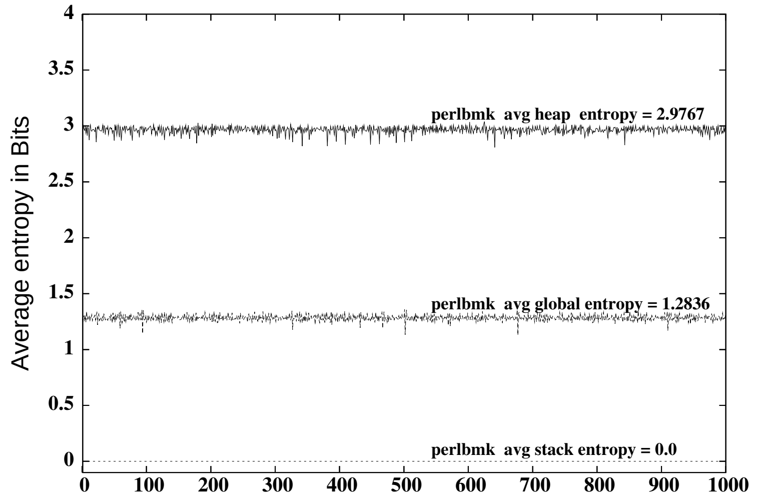
$$H_1 = -\sum_{i=1}^{N} p(x_i) \sum_{j=1}^{N} p(x_j/x_i)\ log_2\ p(x_j/x_i)\ ...\ (2)$$

In general, an nth-order Markov source entropy is a measure of unpredictability given the knowledge of the previous n-1 references. The entropy is zero, when the VPNs generated by the processor are deterministic or can be predicted correctly. The entropy is maximum when the probabilities are equal for all the VPNs.

5.1.1 VPN entropy measurement



(a) Mibench patricia



(b) SPEC 2000 perlbmk

Figure 23: Entropy content of VPNs.

Figure 23 shows the measured zeroth-order Markov source entropy content of the virtual page numbers based on the stack, global static, and heap regions of the patricia program from the MiBench [58] benchmark and the perlbmk program from the SPEC benchmark using equation (1). Each point on the x-axis corresponds to 10,000 VPNs and the corresponding y-axis value is the average entropy content for those 10,000 VPNs.

The entropies of the stack VPNs (the bottom plots) for these two benchmarks are negligible. Hence, there is little or no change in the stack VPNs. The stack region shows low entropy, as the memory accesses to this region happen in an orderly fashion. A good example of this behavior happens during function calls, where the stack frame increases and decreases linearly and the memory references to the stack are close to each other during the execution lifetime of that function. The size of the stack frame depends on the function's activation record.

The middle plot in both Figure 23(a) and Figure 23(b) shows the entropy for the global VPNs. The global data size is fixed based on the global variables declared in the program. For example, 16KB of global data corresponds to four pages based on a 4KB page size. The 20-bit global VPN activity in the plot comes from the accesses within this global data size. The accesses to the global region by a program are more random compared to the stack region accesses, showing that they are a bit more difficult to predict than the accesses to the stack region. An important aspect to note is that, only a few least significant bits out of the 20-bit global VPN vary during the entire program execution. The remaining higher-order bits do not vary. In the above example, only the last two bits (four pages) of the VPN changed and the remaining higher-order 18 VPN bits did not change. Based on this fact, the entropy of the higher-order global VPN bits is zero, meaning, it is deterministic throughout the program execution. As the sizes of various sections are clearly defined in the section headers of the executable file, the global data size is determined during the program compilation. Therefore, the number of pages occupied by the global data can be determined prior to a program's execution.

The top most plot in both Figure 23(a) and Figure 23(b) shows the entropy for the heap region. It has the highest variation among all the regions, indicating that it is the most difficult one to predict, as each VPN contains more information. The entropy of the heap region VPNs is much higher than the stack and global regions, as accesses to dynamically allocated objects (through function calls such as malloc) cannot be tracked easily. The global and heap entropy plot is shorter in length for the Mibench patricia compared to the SPEC2000 perlbmk program, because of fewer memory references. Also, it can be inferred that patricia program has more global references compared to heap, as the heap plot is shorter in length.

5.1.2 Alternative VPN entropy measurement using GZIP

Another way to determine the entropy content is by measuring the compression ratio of the VPNs of the semantic regions, using a standard compression program like gzip. The gzip utility uses the Lempel-Ziv-Welch (LZW) algorithm [108], a universal algorithm for sequential data compression. A universal algorithm is a coding scheme, in which the coding process is interlaced with the learning process for varying source characteristics. One of the main advantages of this algorithm is, it does not need any probability knowledge for the source symbols *a priori*, in our case the VPNs, unlike Huffman encoding [64]. The LZW algorithm relies on the recurrence of strings in its input. Instead of assigning codewords to the symbols in advance, it assigns codewords to repeating source words or patterns in the text. The method is still the basis for many lossless modern data compression techniques. Hence, the size of the compressed trace file compared to the original one gives a good indication of the entropy contained in the VPNs.

Figure 24 shows the uncompressed VPN trace file size in megabytes. The trace file contains the 20-bit VPNs that occurred during the dynamic program execution. The empty bars, such as blowfish global, indicate that the file size is small. Also, Figure 24 shows that many benchmark programs have higher number of stack references compared to the global and heap.

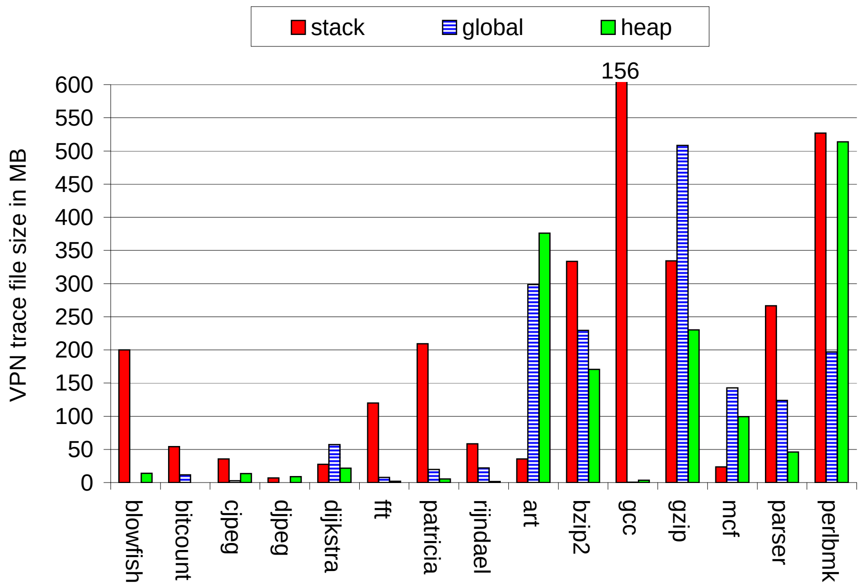


Figure 24: VPN file size before compression.

Figure 25 shows the percentage of the compressed VPN trace file size to the original file size for the Mibench and the SPEC2000 benchmark programs. The standard gzip utility that uses the LZW algorithm [108] is used to compress the trace file. For example, the compressed stack VPN file size for blowfish is only 0.3% of the original VPN file size. The stack VPN has the lowest compression ratio of all the three regions, indicating that the gzip algorithm was able to find the repeating sequence much more often in the stack VPNs than in the global and heap VPNs.

From Figures 21, 22, and 23, we deduce that the stack VPNs have the lowest entropy, in spite of the total number of memory references to the stack being much higher than that to the global and heap for most of the benchmark programs. One outstanding example of this behavior is shown by the gcc program in Figure 24 and Figure 25. Though the stack accesses dominate the gcc program, as shown

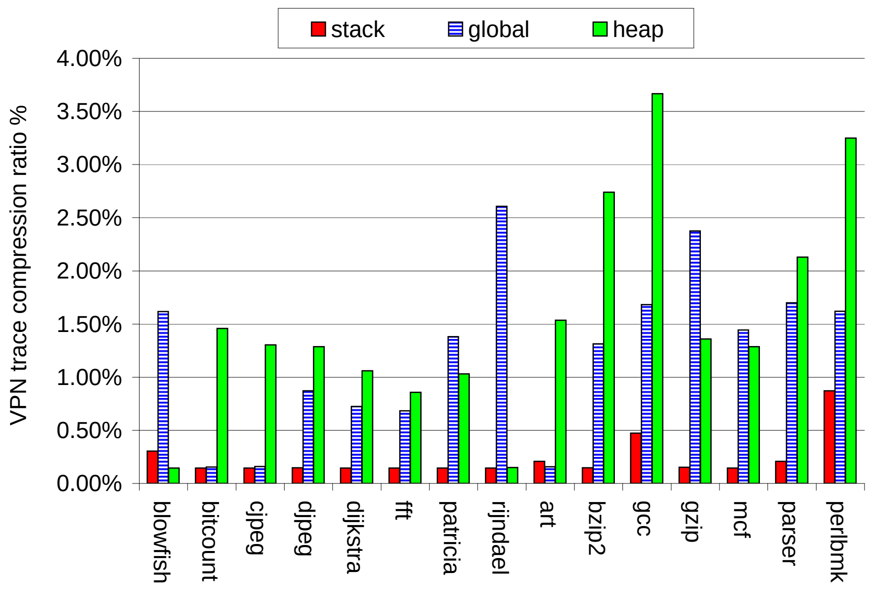


Figure 25: Compression ratio percentage of VPNs.

in Figure 24, its compression ratio is only 0.5%, as shown in Figure 25, indicating its low information content.

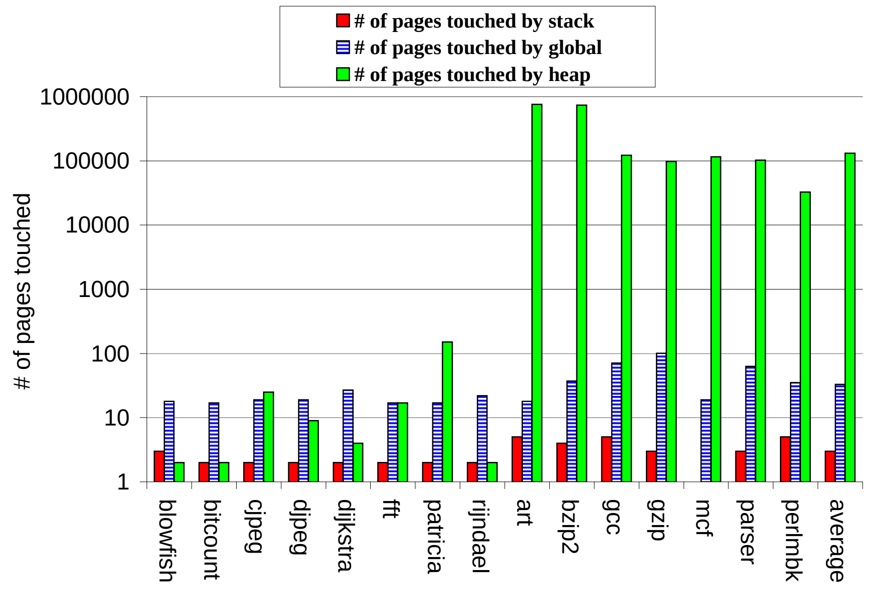


Figure 26: Total number of pages accessed.

Figure 26 shows the number of pages touched by the program from its respective base during the program execution. Sometimes the program dynamically allocates memory (through function calls like malloc) in the heap region and may not utilize all the pages. In such cases, though the virtual page is covered by the program, it may

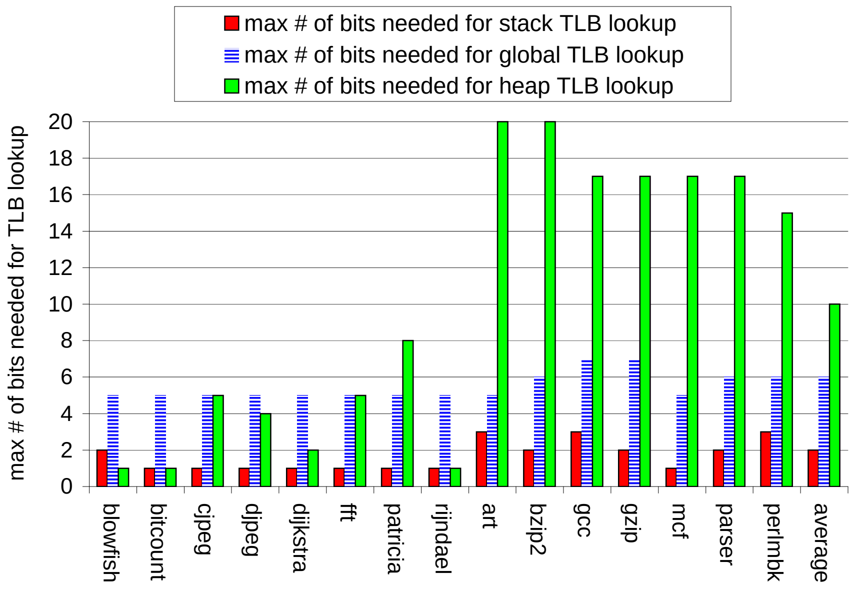


Figure 27: Max number of bits needed.

not have accessed any memory location in that non-utilized page. This is particularly conspicuous for the heap regions, where the memory locations are not sequentially accessed. For example, a program may allocate ten-thousand quad words (around 156 pages) using malloc() function expecting that in runtime this entire memory will be used. But sometimes, in reality, all of them may not be accessed by the program, or possibly only some middle or top few pages are accessed based on some condition checks. For example, in Figure 26, the patricia program covered 151 pages and most of the SPEC benchmarks covered thousands of unique pages during the program execution. This random access behavior makes it difficult to predict, as shown by the high entropy value in Figure 23.

Figure 27 shows the maximum number of bits needed for the entropy-based address translation based on the number of pages covered. The number of TLB bits needed for the entropy-based address translation resolution is directly proportional to the number of pages accessed. The small bars for the stack and global region in Figure 27 suggest that a small number of bits is sufficient for the TLB tag match, instead of the complete 20-bit VPN during the address translation. For example, if the stack region covered eight pages from its base, it requires only three bits for the tag match to resolve the address translation correctly. For most of the SPEC benchmarks, all 20-bits of VPN are required to resolve the heap address translation correctly.

An important characteristic of the global region is, its data size (the total number of pages shown) is deterministic at compile-time. So, unlike the stack and heap regions, the global data size is fixed throughout the program execution and does not change.

In summary, we observe the following characteristics during the program execution.

- The stack VPNs have low entropy and are highly predictable.
- The global data size is determined at compile-time. The number of bits actually needed for the global TLB tag match is proportional to the number of global data pages.
- The number of bits needed for the stack and global address translation tag match is less than the complete 20-bit VPN.

5.2 Entropy-Based Data TLB

Based on the above discussion, it is clear that a few lower-order VPN bits are sufficient instead of the full 20-bit VPN tag match during the TLB lookup for the stack and global memory references. Figure 28 shows a semantic-aware memory (SAM) architecture enhanced with Entropy-based SPeculative (ESP-TLB) and Entropy-based DeTerministic (EDT-TLB) mechanisms. The Entropy-based SAM (ESAM) architecture exploits the characteristics demonstrated in each semantic region by reorganizing the TLBs. The first level TLB structure is reorganized into two small structures, namely, the stack and global static micro-TLBs, and the second level TLB for all the other data accesses including the ones that miss the first level TLBs. The virtual address

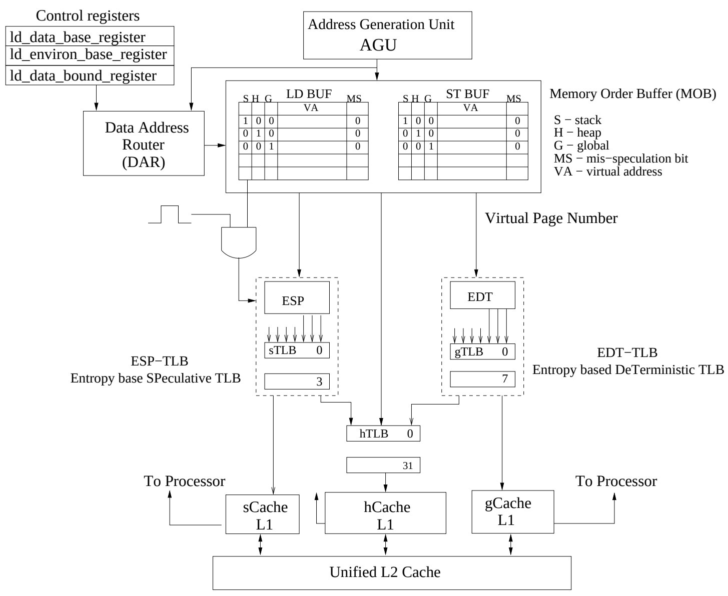


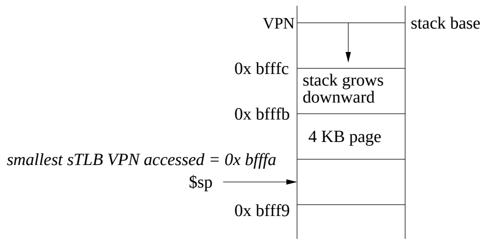
Figure 28: Entropy-based SAM microarchitecture (ESAM).

(32-bits) from the address generation unit (AGU) is written to the the load/store buffer. In the same cycle, the virtual address from the AGU also enters the Data Address Router (DAR). The DAR writes three-bit (100 for stack, 010 for heap, 001 for global) information to the corresponding load/store buffer based on the ranges derived from the control registers (the ld_data_base_register, the ld_environ_base_register, and the ld_data_bound_register) initialized by the system loader. Alternatively, the compiler can annotate the loads and stores with the semantic region information. During the instruction dispatch by the reservation station, the information can be written to the load/store buffer. The MOB uses the three bits (100-stack, 010-heap, 001-global) during the virtual address dispatch and clock gates the other two semantic TLBs that do not participate in the address translation to reduce the dynamic power as shown in Figure 28. The addition of four bits (three bits - one each for the stack, global, and heap, and one bit for mis-speculation) to the load/store buffer is not a significant overhead, as it already contains many bits like the page fault information, data size of the load/store, load/store color ids, etc. Though it is not a significant overhead, we account the power consumed by the four bits in our simulation. Also, as the four bits do not participate in any kind of associative search, the power consumed is less. The ESAM speculatively translates the stack addresses using the ESP-TLB structure and deterministically translates the global static address using the EDT-TLB structure. The implementation details of the two structures are described in the following sections.

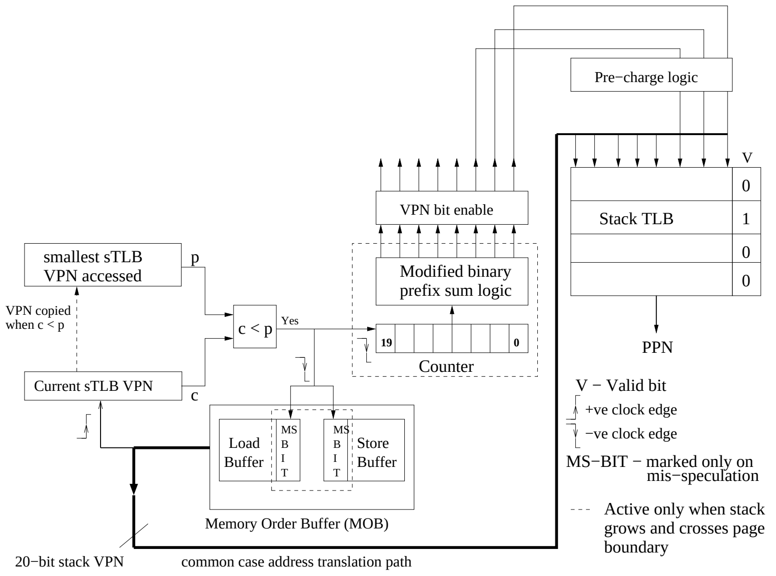
5.2.1 Entropy-Based SPeculative TLB

As we have shown previously, the stack addresses have low entropy and are highly predictable. To utilize the characteristics, stack addresses are speculatively translated. Figure 29 shows the block diagram of the ESP-TLB. At an higher level, the logic block first speculatively pre-charges the same number of bits that was used in the previous cycle VPN translation for the current VPN translation and evaluates its correctness. If it finds that it is incorrect, the logic block does three things: 1) corrects itself by increasing the number of pre-charge bits, 2) squashes the incorrect translation by invalidating the cache tag match, and 3) finds the correct translation in the subsequent cycle.

The main functionality of the logic block is to enable and pre-charge the minimum required number of bits for the stack TLB (sTLB) address translation to save the energy consumption in the TLBs. The *smallest sTLB VPN accessed* register denoted "p" is initialized to 0xffffffff. In our simulation model based on the ARM architecture, the stack region begins from higher memory address and grows downward. During the dynamic program execution, the register will hold the smallest VPN that was accessed by the stack pointer ($sp). The smallest VPN should represent the memory page pointed by the current $sp, based on the software convention. Any memory access beyond the current $sp is considered as a violation of the software convention. The counter bits, mis-speculation bit (MS-BIT) in the load/store buffer, and the TLB



(a) Stack growth in ARM memory model


Pre−charge logic
V
Stack TLB
0
1
0
0
PPN
VPN bit enable
Modified binary prefix sum logic
smallest sTLB VPN accessed
p
VPN copied when c < p
c < p
Yes
19
0
Counter
Current sTLB VPN
c
Load Buffer
MS B I T
MS B I T
Store Buffer
Memory Order Buffer (MOB)
20−bit stack VPN
common case address translation path
V − Valid bit
+ve clock edge
−ve clock edge
MS−BIT − marked only on mis−speculation
Active only when stack grows and crosses page boundary

(b) Entropy-based SPeculative stack TLB

Figure 29: ESP-TLB.

valid bits (V) in each entry are initialized to zero. The MS-BIT is set/reset based on the correctness of the speculatively translated VPN. Figure 29(a) shows the stack growth in the ARM memory model. As the stack grows, the VPN decreases, visually represented in Figure 29(a).

The memory order buffer (MOB) dispatches the virtual address that needs to be translated to the semantic TLBs. But, only one TLB is enabled based on the three-bit information stored as part of the load/store buffer. If the sTLB is enabled, the sTLB is pre-charged only for the bits that are high at the output of the *VPN bit enable* register, a bit mask. The VPN bits sent from the load/store buffer to the stack TLB represent the common case for stack address translation. The common case path is shown in bold line from the load/store buffer to the stack TLB in Figure 29(b).

At the same time the address reaches the sTLB, the *current sTLB VPN* denoted "c" that contains the current VPN is compared with the *smallest sTLB VPN accessed* during the high phase of a clock cycle. If the current stack reference did not cross the page boundary, the comparator output will be FALSE. When the comparator output is FALSE ($c \geq p$), the speculative translation was indeed correct and no further correction is needed. The counter value stays the same and is not decremented. Note that the comparison is not on the critical path of the stack address translation.

When the stack region grows (downward), as shown in Figure 29(a), and crosses the page boundary, the comparator output will be TRUE ($c < p$), indicating that the speculatively translated address was incorrect. Hence, a mis-speculation recovery is required that includes the following steps: 1) sets the mis-speculation bit (MS-BIT) in the memory order buffer for this virtual address entry, 2) updates the *smallest sTLB VPN accessed* register with the *current sTLB VPN* value, 3) increments the counter, and 4) invalidates the output of the cache tag match hit/miss signal by qualifying it with the comparator output signal. All four events happen on the negative edge of the same cycle as there is no dependency between them.

The counter value represents the number of pages the stack region has grown and covered from the stack base. The *modified binary prefix sum logic* is a combinational circuit that determines the minimum number of the sTLB pre-charge bits to be enabled using the counter value. The output of the *modified binary prefix sum logic* is stored in the *VPN bit enable* register and is ready for the correct address translation in the subsequent cycle. The incorrectly translated address and data from the cache will get squashed by an invalidation of the cache tag match, based on the same signal from the output of the comparator that writes the MS-BIT. The number of mis-speculations for the stack TLB is much less than the number of predictions. The corresponding memory reference is re-issued by the MOB scheduler and the MS-BIT is reset after the re-issue. The re-issued memory reference succeeds through the common case path this time, as both the *current sTLB VPN* and *smallest sTLB VPN accessed* contain the same value and the output of the comparator will be FALSE this time. The dependent instructions on the load data will get woken up and dispatched to their respective execution units. The power consumption of all the components in the Figure 29, the number of mis-speculations, and the performance issues on the mis-speculation path are discussed in Section 7.8.

The load/store buffer is extended by the addition of the MS-BIT for each entry, as shown in the Figure 29. Under normal operation, the MOB conveys the information to the load/store buffer during the TLB page-faults and memory-access violations, and reschedules the affected memory operations after the TLB page walk. The existing MOB mechanism can be extended to set and reset the MS-BIT and no further complex logic is required.

The *modified binary prefix sum logic* is a combinational circuit that determines the minimum number of the sTLB pre-charge bits to be enabled based on the counter value. For example, if the program has covered 12 stack pages from the stack base, the counter value will be 12 (binary 1100). The 20-bit VPNs of these 12 pages can be uniquely identified using the four lower-order bits [3:0], as the stack frame grows sequentially. When a new VPN lookup the sTLB for address translation, these four bits are enough to resolve the 12 pages unambiguously for the VPN tag match. Therefore, the sTLB needs to be pre-charged with only the four least significant bits enabled, which are [000....1111]. The logic that takes binary [000...1100] from the 20-bit counter and gives a 20-bit binary [000...1111] is achieved using a binary prefix sum logic. In general, the mathematical representation of the binary prefix sum logic is, given an n-bit binary input sequence,

$b_{n-1}, b_{n-2}, ...b_2, b_1, b_0$

the n-bit binary output sequence is,

$p_{n-1}, p_{n-2}, ...p_2, p_1, p_0$

where $p_i = b_{n-1} + b_{n-2} + ... + b_{i+1} + bi$ is called as the i^{th} binary prefix sum [80].

We need to take care of an additional case, for example, when the counter value is four (binary 100), indicating four stack pages are now covered by the program, the output should be binary 011, as two bits are enough to uniquely identify these four VPNs. But the output of the binary prefix sum logic will be 111. This pre-charges one extra bit that is not needed for the address resolution, consuming more energy during every stack TLB lookup. Therefore, whenever the counter value is a multiple power of two, the MSB of the binary prefix sum logic output must be changed to zero instead of one. We modify the truth table accordingly to accommodate the above case and implement the logic using simple gates. The *modified prefix sum logic* enables pre-charging only the few least significant bits of the stack TLB, saving energy by avoiding the complete 20-bit VPN pre-charge. The power consumption and the delay for this logic are discussed in the experiments section.

The following things are to be noted during the stack address translation mechanism:

1. The stack address is translated speculatively through the common-case (highlighted) path shown in Figure 29.

2. The comparator, the modified binary prefix sum logic, and the MS-BIT update in the load/store buffer do not affect the critical path of the TLB lookup, as the stack address translation is speculative.

3. The modified binary prefix sum combinational logic becomes active and consumes power only on mis-speculation.

5.2.2 Entropy-based DeTerministic TLB

The compiler, assembler, and linker tool chain compiles the program and creates a widely-used ELF or COFF file format that is loadable, relocatable, and executable.

As part of the process, the global variables in the program become part of the global static data in the executable. The size of the .text, .init, .data, .rodata, and the .bss sections are clearly defined as part of the executable file format. Therefore, the number of pages occupied by the global static data can be determined after the program compilation from the difference between the loader variables *ld_data_start* (the address where the global static data starts) and the *ld_data_bound* (the address where the global static data ends). Once the global data size is known, the maximum number of pages occupied by the global data is obtained by dividing the global data size by 4096 (the page size). The minimum number of bits required to address the global pages and the bit sequence to the pre-charge logic to enable the global TLB are calculated by the loader program, as shown below. By going through a few lines of code once, the loader can deterministically find the bits that are needed to pre-charge and stores them in the *deterministic fixed bits* control register.

global data size = ld_data_bound - ld_data_base;

number of pages = global data size / 4096;

number of bits needed = log_2*(number of pages);*

deterministic fixed bits = mod_binary_prefix_sum(number of bits needed);

The last line is a software equivalent of finding the modified binary prefix sum bit sequence that was used in Section 5.2.1.

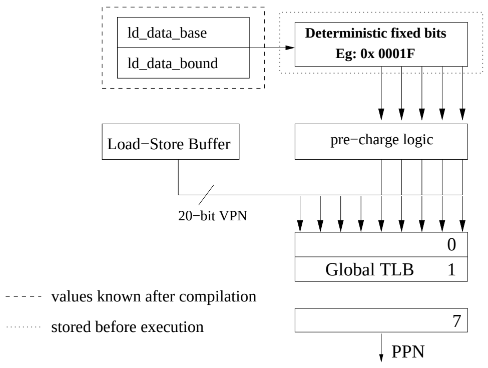


Figure 30: Entropy-based DeTerministic (EDT) global TLB.

The global entropy activity comes from the memory references between the defined

Table 11: Processor Configuration.

32-bit Processor Parameters	**Values**
Execution Engine	4-wide out-of-order
Number of data TLB entries	32
Page size	4 KB
L1/L2 cache hit latency	1 / 6 cycle
Memory latency	150 cycles
TLB hit/miss latency	1 / 30 cycles
L1 Cache baseline	DM, 32KB, 32B line
L2 Cache	4-way 512KB, 32B line
Number of TLB ports used	1
20-bit comparator power	300μW
Modified prefix sum logic delay	160 ps
Modified prefix sum logic power	668 μW
Dynamic power for counter	956 μW

boundaries of the global data size. As the activity is confined to these few pages occupied by the global data size, only a few lower-order bits change between memory references and most of the higher-order bits of the VPN remain unchanged. It creates an opportunity to enable and pre-charge only the few lower-order bits for the gTLB lookup in order to save energy. In Figure 30 the *deterministic fixed bits* is derived from the loader variables and stored in the control register. The gTLB is pre-charged only for the bits that are high in this register to save energy. As the pre-charge bits are deterministically found before program execution, there is no performance penalty involved.

5.3 Experimental results

Our performance evaluation infrastructure is based on the Simplescalar simulator for the ARM ISA. We integrated the Wattch [38] power model with the Simplescalar ARM simulator for power simulation and modified the Simplescalar to model the ESP-TLB and the EDT-TLB. We use 100nm technology in our simulations. We simulate the Mibench benchmark programs to the end. The SPEC2000 simulations are run for 300 million instructions after fast-forwarding the first one billion instructions. Table 11 describes our machine configuration and power data.

We simulated the logic components using HSPICE. On mis-speculation, the energy consumed by the comparators, the counters, and the modified binary prefix sum logic are added. We charge two penalty cycles for each stack address mis-speculation. On the positive edge of the first cycle, the VPNs are compared. On the negative edge of the first cycle, if the speculative stack address translation is incorrect, the counter is incremented and the modified binary prefix sum logic changes its output. In the second cycle, the MOB reschedules the virtual address for the sTLB lookup. We also add an extra 10% [96] of the dynamic power to account for the leakage power, when there is no TLB access activity.

Figure 31 shows the energy savings using the ESP-TLB and the EDT-TLB mechanisms. The baseline in the figure is a conventional 32-entry d-TLB. The total energy savings attained is 47% with a performance penalty of less than 1%. The penalty from the stack address translation mis-speculation is negligible, as the number of mis-speculations is few. In some cases, there is a performance improvement as the ESAM architecture reduced the number of expensive TLB misses.

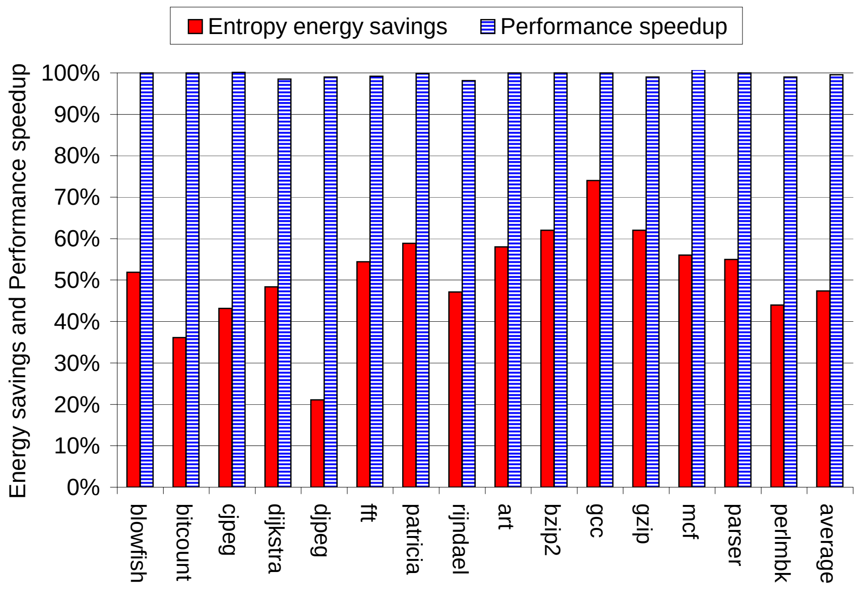


Figure 31: Energy savings and performance using the ESP-TLB and the EDT-TLB.

Figure 32 shows the effectiveness of the ESP-TLB. The maximum number of bits needed to pre-charge for the stack address translation is three bits at most, as the stack region did not cross more than eight page boundaries. The maximum number of mis-speculations is three, because the counter gets incremented from zero to one, one to two, and finally from two to three. The three bars in Figure 32 shows the number of accesses when the sTLB lookup needed to pre-charge one bit, two bits, and three bits, respectively. The experimental results confirm the graphs in the motivation section that the stack accesses have low entropy, are predictable, and can be used to save the energy.

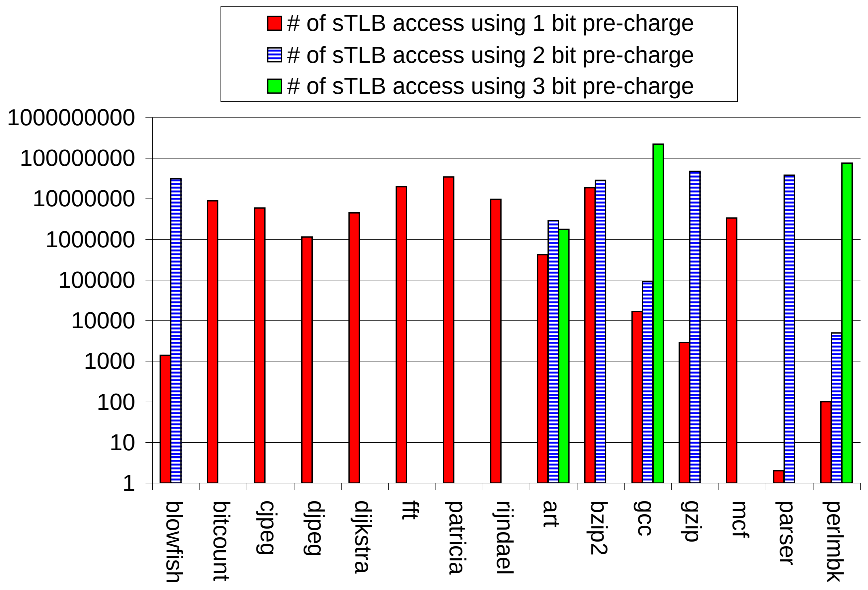


Figure 32: Effectiveness of the ESP-TLB.

5.4 Summary

We quantified the entropy content of the VPNs for the data TLB lookups, and proposed two complexity effective and energy efficient architectural techniques by exploiting the low entropy of the stack and global data. The energy savings are achieved via: 1) the ESP-TLB, a speculative, but highly accurate stack address translation, and 2) the EDT-TLB, a deterministic global static address translation. In both the schemes, we precisely enabled and pre-charged only the minimum number of least significant address bits required to reduce the data TLB energy. We showed that the TLB energy consumption is reduced by 47% with less than 1% performance loss. As shown in our results, the performance degradation because of mis-speculation is negligible for the stack TLB. There is no performance penalty for the global TLB translation, as it is deterministic. As the TLB power continues to rise and we migrate from a 32-bit to a 64-bit architecture supporting large page sizes, our techniques will become more significant in reducing the data TLB energy.

CHAPTER VI

ENERGY EFFICIENT TLB DESIGN FOR JVM

6.1 Introduction

The concept of “write once, run anywhere” has made the Java platforms and their applications widely adopted from wireless hand-held devices to high performance servers. The applications being incorporated into these platforms are also becoming more sophisticated and diversified, as a result, consuming more energy. To extend the battery lifetime and reduce the cost of cooling solutions, energy efficiency is no longer a desired feature but a design constraint. To address the overall energy consumption, it becomes inevitable that a designer needs to consider energy efficiency in all different design stacks including circuits, microarchitecture, compilers, and even programming languages.

The Java language combines different features from different programming paradigms. The features include portability, object-oriented design, multithreading, garbage collection, and exception handling. These rich features come at the expense of decreased performance caused by the additional hardware abstractions. The Java source code is first translated into the architecture-independent bytecodes. These bytecodes can then be executed on any platform that supports an implementation of the Java Virtual Machine (JVM). The JVM insulates the Java application from any contact with the underlying hardware. The JVM is an abstract computer implemented on top of a real hardware and operating system to run compiled Java bytecodes.

A JVM can be implemented via the following mechanisms: an interpreter, a Just-In-Time (JIT) compiler, a dynamic compiler, and a direct hardware execution. There are pros and cons with respect to performance and power to each of these techniques.

The interaction between the Java program and the JVM for the shared resources also influence the power and performance of the overall system. An interpreter emulates the virtual machine by continuously fetching, decoding, and executing the bytecode until the program completes. Although the size of the interpreter is typically small (within hundreds of kilobytes), the per-bytecode translation results in low performance. A JIT or a dynamic compiler compiles the bytecodes into the corresponding machine-specific binaries to execute them natively on the machine. Further, the native code can be dynamically optimized using runtime feedback profile with a dynamic compiler. The JIT methodology improves the performance of the application. But the performance also depends on the effectiveness of the JIT compiler itself. Today, a sophisticated compiler needs more than hundreds of kilobytes and memory space to translate and optimize the code for higher application performance. The last option is to run the bytecodes directly on a Java processor. This eliminates the abstraction of the software translation for the best possible performance. We concentrate on the JIT and dynamic compiler approaches for their prevalence in practice.

The Java programming language offers features such as strong type checking and dynamic garbage collection with the extra layer of the JVM implementation that impacts both power and performance. In the embedded domain where Java is widely deployed with virtual memory support, power efficiency becomes even more important. Prior studies have shown that the memory behavior of the Java applications is very different from the regular C/C++ programs [74, 91, 54, 101]. In particular, the characteristics become very interesting when the compiler, optimizer, and garbage collector of a JVM interacts with the Java bytecodes in the same address space. The instruction TLB and the data TLB are accessed by the JVM code, its associated data, and also by the Java application code and data. Due to the interference of these accesses, the TLB power and performance of Java applications are unexpectedly exacerbated.

6.2 Motivation

To gain better understandings regarding the interaction of memory accesses of the JVM and the Java applications, we briefly discuss the way JVM loads a Java program and executes it in the interpreter, JIT, and dynamic compilation mode.

6.2.1 Interpreter Mode

In the interpreter mode, the JVM first loads the Java class in the dynamic heap memory. The JVM will then fetch, decode, and execute the Java application bytecode that flows through the dTLB and data cache (dCache). So, the dTLB and dCache are accessed by both the JVM's private data and the Java application bytecodes.

6.2.2 Just-In-Time and Dynamic Compilation Mode

In the Just-In-Time mode, the JVM first loads the Java class into the heap memory. A JVM compiler first compiles the bytecode to the native machine binaries of the target processor. In the dynamic compilation mode, the compiler may initially generate an unoptimized code. At runtime, the dynamic optimizer will continuously optimize the binaries with dynamic information. In both modes, the compiled code is stored in the heap space, which will always access the dTLB and dCache. In addition, the JIT or dynamic compiler that optimizes the code during runtime will also go through the dTLB and dCache as part of its own private data accesses.

Once the optimized/unoptimized code is written to the heap through dTLB and dCache, the native code on the heap will be executed through the iTLB and the instruction cache (iCache), so does the code of the JVM itself. This behavior is quite different from the normal execution of a C/C++ program, where instructions are only accessed within the code space. In the JIT or dynamic compilation mode, the JVM code will access the heap memory to fetch the Java application's code. As such, the iTLB is accessed by both the JVM and the Java application. This cross-sharing creates interference in the iTLB, increasing the iTLB miss rate, as the heap

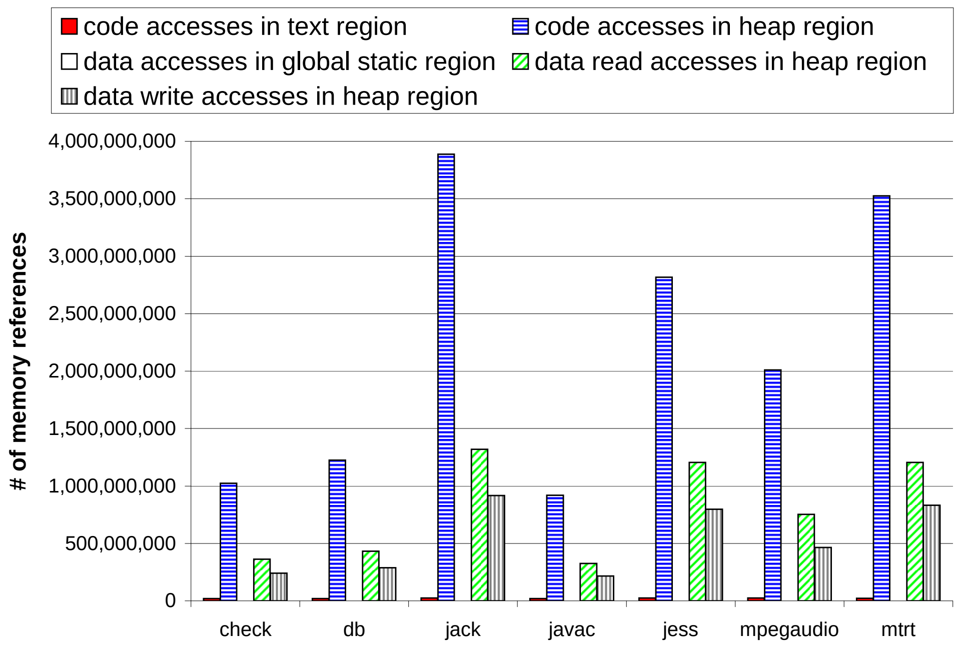


Figure 33: Distribution of instruction and data memory references to different memory regions by the Java application running on a JVM

accesses for Java applications are dynamic and active. The activity is even busier in the dynamic compiler mode since the dynamic JVM optimizer will attempt to optimize the code at runtime while these accesses will potentially conflict with its own instructions in the iTLB and iCache, penalizing performance of both the Java applications and the JVM itself.

Figure 33 shows the distribution of instruction and data accesses to different regions of the virtual address space for the seven SPECjvm98 [22] benchmark programs using input set s1. The distribution information was collected using the Dynamic SimpleScalar [9, 63] simulator simulating the Jikes RVM [16] running the Java applications. We modified the Dynamic SimpleScalar simulator to identify the regions of memory accesses and collect the distribution profile.

In this figure, the first two bars are plotted for the instruction side while the rest of the three are plotted for the data side. The leftmost bar shows the number of memory accesses to the code region followed by a bar showing the number of instruction accesses to the heap. The JIT compiler generates and writes the machine-dependent native instructions to the heap. These native instructions are read from the heap to execute the Java application. Notice that these heap accesses are dominant in the instruction side, accounting for almost all the instructions accessed.

The next three bars show the distribution of data memory accesses. The third bar shows the number of accesses to the static global region, which is apparently insignificant. The last two bars show the numbers of "data" reads from and writes to the heap. As mentioned earlier, part of these accesses are caused by the activities that the JIT compiler reads the Java bytecode and generates native code that is written onto the heap. We will exploit these memory access characteristics between the JVM and the Java applications for achieving an energy-efficient TLB design.

6.3 JVM and Java iTLB (J-iTLB)

As indicated in the second bar of Figure 33, most of the instructions that go through the iTLB are in fact coming from the heap. These heap accesses are to fetch the native code and execute it. Unfortunately, these highly active accesses can adversely conflict with the normal JVM's code accesses, increasing the iTLB misses in the traditional iTLB structure. To address this interference issue, we propose a new structure that segregates the iTLB into two distinct TLB structures to improve both performance and power. Using two distinct iTLBs, the JVM code and the JIT-compiled application code will be stored separately to eliminate the iTLB interference completely. Also, as the JVM code itself is static, the requirement of its iTLB can be designed much smaller than one holding the JVM-compiled code. On the other hand, Java application codes are much more dynamic since the objects are created and freed at runtime in the Java programs. Worse yet, the unused or deallocated objects can be highly fragmented before they are recycled by the garbage collector. As a result, new objects will be less likely to be allocated within the same memory page or in consecutive pages, causing more TLB misses.

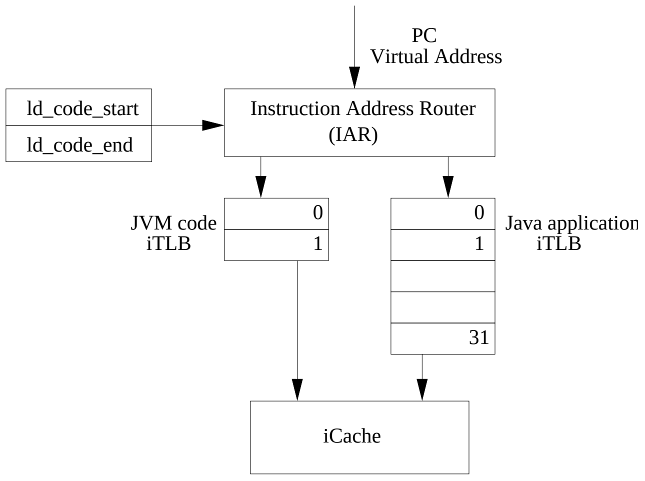


Figure 34: JVM and Java iTLB (J-iTLB)

Figure 34 illustrates our proposed iTLB organization in which the iTLB is horizontally split into two TLBs: one dedicated for the JVM code accesses, and the other for the Java application accesses to the heap. Not only does such segregation eliminate the conflict misses of address translations from two unrelated code space, it also improves power consumption when only one of the TLBs is looked up. During the program's start, the loader identifies the ld_code_start address (code start) and ld_code_end (code end) of the JVM. This information is readily available as part of the ELF or COFF executable format. The sizes of various sections such as .text, .init, .data, .rodata, and .bss sections are clearly defined in the executable file. These two addresses are then kept by a special hardware register pair stored inside the Instruction Address Router (IAR) as shown in the figure. The IAR will route each of the incoming PC virtual address based on this register pair. All incoming PC virtual addresses are checked against the register pair. If an address falls in the JVM's text range, it is routed to the JVM iTLB; otherwise, it is forwarded to the Java application iTLB. Also, the outcome of these comparisons by the IAR is used to clock-gate the unused iTLB for power savings.

A typical Java program creates many objects interacting with other objects using

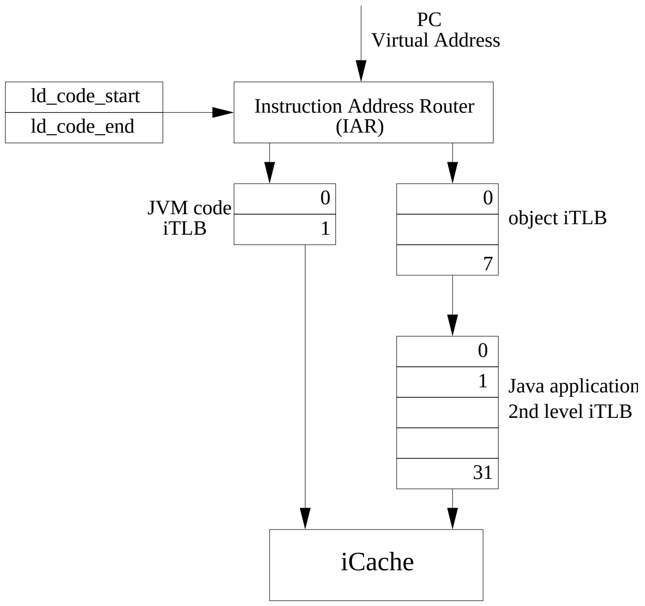


Figure 35: JVM and Java iTLB (J-iTLB) with object iTLB

the methods. Once the object completes the work, its resources are released and re-allocated for other objects. These objects generally access both code and data memory regions, and are typically small with good locality and short life span [71]. In fact, these short-lived objects constitute a high percentage of the total memory references. Given the Java objects are small and dynamic, we introduce an object iTLB to exploit this behavior for reducing power.

Figure 35 shows our proposed iTLB organization. In addition to the partitioned scheme shown in Figure 34, the iTLB for the Java application code is further stratified into two levels. The first level, a smaller TLB, is inserted to support the dynamic object code accesses in the Java applications. It is then backed up by a second-level, larger TLB to improve the TLB miss rate. Again, the IAR routes the addresses based on the virtual address as explained earlier. We charge an additional extra cycle when it misses the object iTLB and looks up the second-level iTLB. We will analyze the power and performance of these two iTLB organizations in Section 7.8.

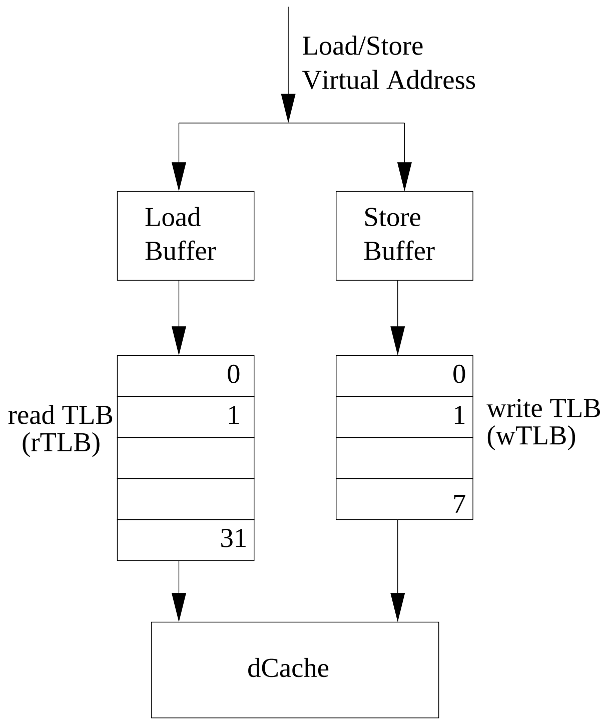


Figure 36: JVM and Java dTLB (J-dTLB)

6.4 JVM and Java dTLB (J-dTLB)

As shown in the last three bars of Figure 33, the numbers of reads and writes to the heap account for the majority of data memory accesses in Java applications, dwarfing the number of reads in the global static data region. The JIT compiler translates the Java application bytecode into the native code and writes it in the heap space. These writes flow through the dTLB and the dCache. Finally, the underlying hardware reads the translated native code through the iTLB and the iCache.

To exploit the characteristics of data accesses by Java applications, we propose to modify the dTLB as shown in Figure 36. In this new organization, the dTLB is split into two distinct TLBs, one for reads and one for writes. All load addresses are directed to the read TLB (rTLB) while all store addresses are routed to the write TLB (wTLB) to minimize the contention between them. Also, when the Java application is in execution, the JIT or dynamic compiler uses the wTLB for native code writes

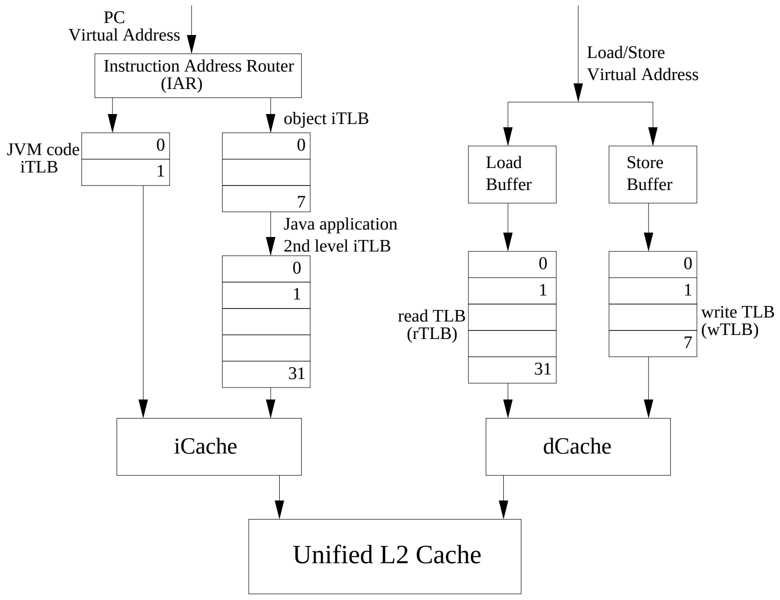


Figure 37: Complete organization of iTLB and dTLB

while the rTLB is used by the JVM and the Java application data accesses to read the data from the heap simultaneously without the need to multi-port a monolithic dTLB. Figure 37 shows the complete schematic of our proposed energy efficient TLB structure for Java applications running on JVM.

6.5 Experimental results

Our power and performance evaluation infrastructure is based on Dynamic Simplescalar (DSS) [9] [63] simulator. The Dynamic Simplescalar simulator simulates Java programs running on a JVM that uses Just-In-Time or dynamic compilation. The Dynamic Simplescalar allows JIT compilers such as Jikes RVM to be simulated on top of it. The DSS also supports many other interesting features such as PowerPC ISA target, checkpointing, a vastly improved memory model, and a Wattch-based power model [39] [49] using 100nm technology. As the DSS simulates PowerPC ISA as the underlying target machine, we used GNU cross compiler tools and utilities to generate the Jikes RVM PowerPC binary, and the RVM code and data images on a

Linux hosted Intel IA-32 machine. The Jikes RVM developed by IBM researchers includes an aggressive optimizing compiler and a flexible dynamic adaptive compilation infrastructure.

The SPECjvm98 benchmark suite [22] consists of eight programs and most of them are derived from real world applications. The SPECjvm98 allows users to evaluate the performance of both the hardware and the software aspects of a JVM platform. On the software side, it evaluates the performance of the JVM, JIT, and the operating system implementations. On the hardware side, it includes the CPU, the caches, and the memory sub-system. We simulated all the Java applications in the SPECjvm98 suite except *compress*, which had problems when running on Dynamic SimpleScalar. We ran all the programs in the SPECjvm98 from start to its completion without fast forwarding or skipping instructions. The simulated processor configuration is shown in Table 15.

32-bit Processor Parameters	**Values**
Execution Engine	in-order
Number of baseline ITLB entries	32
Number of baseline DTLB entries	32
Number of JVM ITLB entries	2
Number of object ITLB entries	8
Number of 2nd level ITLB entries	32
Number of read DTLB entries	32
Number of write DTLB entries	8
Page size	4 KB
L1/L2 cache hit latency	1 / 6 cycle
Memory latency	150 cycles
TLB hit latency	1 cycle
L1I and L1D cache baseline	4-way associative 32KB, 32B line
L2 cache	4-way 512KB, 32B line
Number of TLB ports used	1
Each 20-bit comparator power	300μW

Table 12: Processor model parameters

First, we evaluated the effectiveness of the new J-iTLB structure as proposed in Figure 34. The power savings for this configuration is shown in the leftmost bar of Figure 38. In this experiment, we focus on the iTLB and assume a 32-entry conventional dTLB for all the experiments. The extra energy consumed by the two

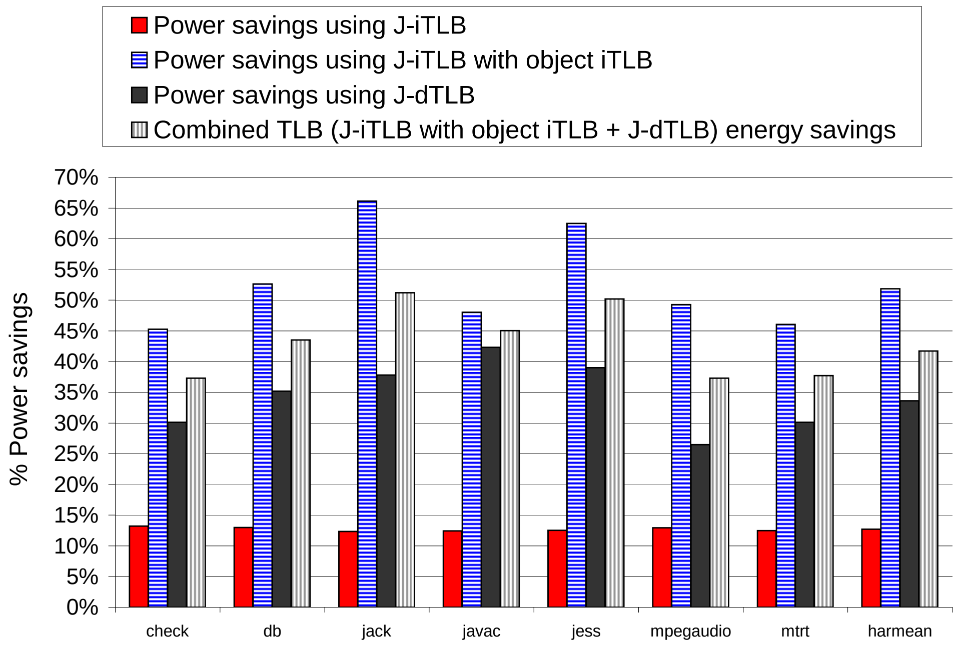


Figure 38: Power savings using J-iTLB, J-iTLB with object iTLB, and J-dTLB

comparators in the Instruction Address Router is taxed every cycle. Each comparator consumes 300μW using 100nm technology based on SPICE modeling. The energy savings is around 12.7% without any performance penalty. In fact, there is slight performance improvement around 1%. The reason is that the interference between the JVM and the Java application accesses is eliminated in the new partitioned TLBs. The performance improvement is shown in the leftmost bar of Figure 39.

The second experiment is based on the TLB configuration depicted in Figure 35, where the J-iTLB is combined with a first level object iTLB. This configuration achieved higher energy savings of 51% as shown in the second bar of Figure 38. We charge an extra cycle for the object iTLB misses that access the second level iTLB. The performance impact is around 1% as shown in the corresponding bar of Figure 39.

The third experiment we conducted evaluates the power and performance impact of the segregated d-TLB configuration proposed in Figure 36. The third bar in Figure 38 shows the power savings achieved using the split dTLB for reads and writes. The power savings is around 34% on average. As the effective number of dTLBs are

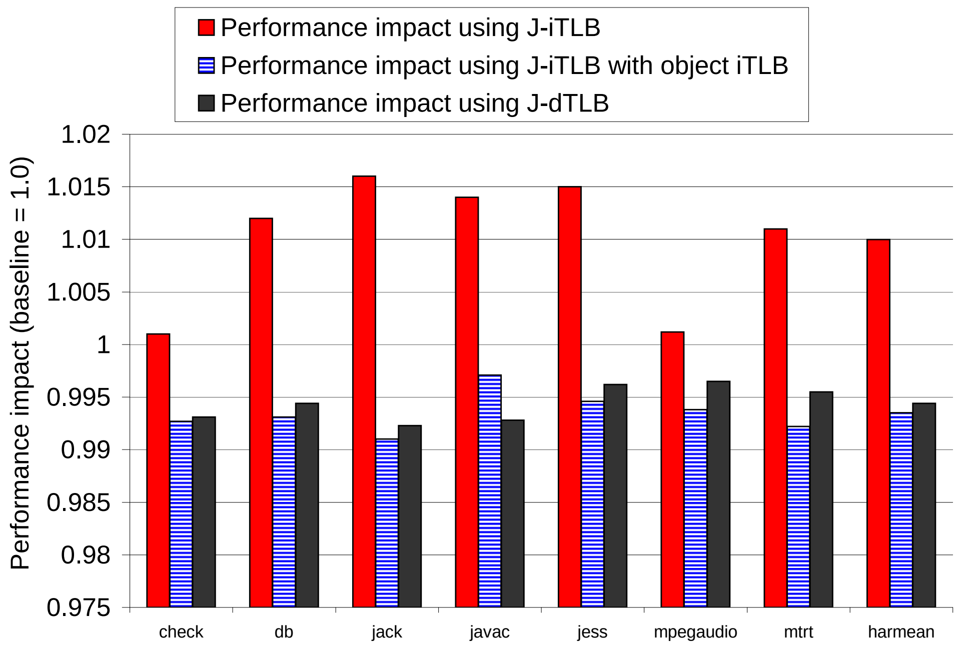


Figure 39: Performance impact of J-iTLB, J-iTLB with object iTLB, and J-dTLB

now divided, they cause around 1% performance impact. The performance impact is shown in the rightmost bar of Figure 39.

We finally combined the J-iTLB with object iTLB configuration and the J-dTLB configuration to evaluate the overall TLB power savings in the processor. In this combined configuration, we obtained 42% overall TLB power savings as shown in the rightmost bar of Figure 38 with less than 1% performance penalty.

6.6 Summary

We proposed an energy efficient Translation Lookaside Buffer design for the Java applications running on JVM without very little performance perturbation. Our new partitioned TLB structures exploit memory reference characteristics and the interactions between the JVM and Java applications. The method horizontally partitions the traditional monolithic iTLB and dTLB into distinct, smaller TLBs for special purposes targeting for Java applications. Our J-iTLB scheme can reduce energy by 12.7% with a performance improvement of 1%. The performance improvement was obtained from the elimination of conflict misses between the JVM code and the Java application accesses in the iTLB. The energy savings is increased to 51% by combining the J-iTLB with a first-level small object iTLB at the expense of 1% performance impact. The J-dTLB organization, where the dTLB is split into two for reads and writes provides an energy savings of 34% with 1% performance impact. When the J-iTLB with object iTLB is combined with J-dTLB, we obtained 42% overall TLB energy savings with less than 1% performance impact. As the TLB power continues to rise, our energy-aware TLB design will become more significant in reducing the TLB energy.

CHAPTER VII

SNOOP POWER REDUCTION IN CHIP MULTIPROCESSOR

7.1 Introduction

The continuous miniaturization of devices by the process technology has brought the on-die homogeneous and heterogeneous multicore processors or chip multiprocessors into all market segments ranging from servers to mobile products. In addition to improving performance, CMP can also be used to address emerging issues such as security [99], reliability [57, 104], 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* CMP in a CMP-SMP system. Prior research work and studies 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) increasing number of cores per die, 2) continuous extension of the vertical cache hierarchy, and 3) increasing size of cache and load/store buffers. 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 requirements to send snoop probes differ based on cache inclusion policies, leading to different response times.

Cache inclusion properties were first studied by Baer and Wang [29] to 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 other inclusion policies. For example, IBM's Power5 [21] uses strongly inclusive caches [29], Intel Pentium Pro uses weakly inclusive caches [82], and COMPAQ Piranha [33] and AMD Athlon [3] use exclusive caches in their design. In exclusive caches, data is in only one of the caches in the hierarchy to increase the effective cache capacity. All three cache policies have pros and cons. 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 that are 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 lower level caches only when a 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 cores in inclusive caches is to add a *shadow* set of all lower level cache tags near the shared last level cache to maintain coherency. Unfortunately, the overhead to maintain these shadow tags will become higher as the number of cores, cache hierarchy, and size of the lower level caches increases.

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

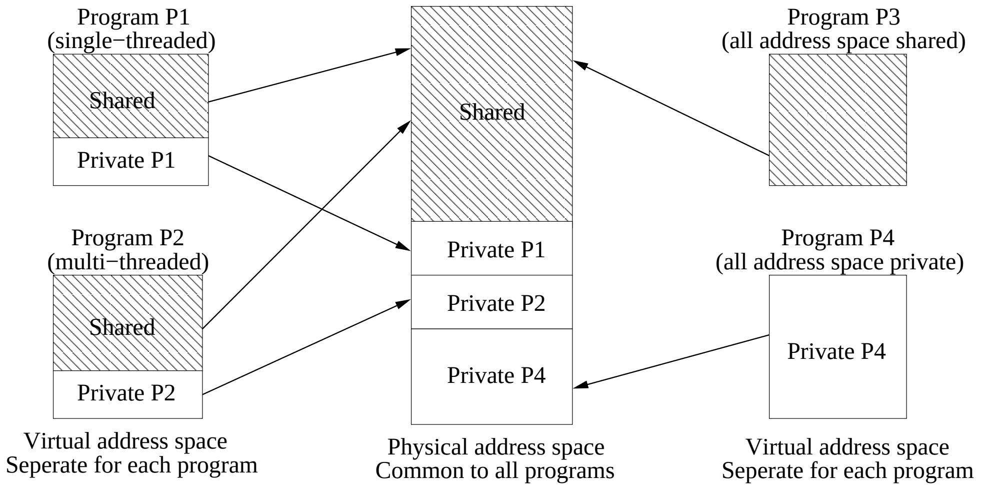


Figure 40: Various program categories.

between single and multi-threaded applications can play an important role in determining the number of snoop probes generated and their behavior. 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 the 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 the 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 the resources they use.

7.2 *Program variables and snoop probes*

Figure 40 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

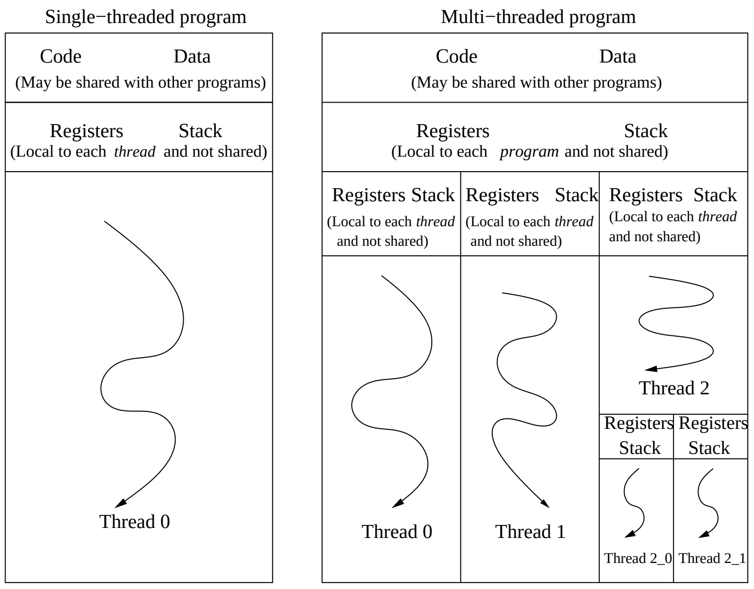

Single−threaded program
Code
Data
(May be shared with other programs)
Registers
Stack
(Local to each thread and not shared)
Thread 0
Multi−threaded program
Code
Data
(May be shared with other programs)
Registers
Stack
(Local to each program and not shared)
Registers Stack
(Local to each thread and not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

Figure 41: Thread stack in single and multi-threaded programs.

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 other programs* in the system. 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 41 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 40. 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 40.

The knowledge about the variables that are shared in a program provides a good opportunity to optimize snoops in a shared memory system environment. Current cache coherence implementations do not differentiate between single and multi-threaded programs that can co-exist in a CMP. Sometimes a single-threaded program may not use any shared memory constructs such as semaphores or locks. 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 proposed a hardware-only technique called Selective Snoop Probe (SSP) that exploited the properties of stack variables to filter out unnecessary snoops. Then we proposed a hybrid hardware/software technique called Essential Snoop Probe (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 techniques can be adopted in all types of inclusive/exclusive cache policies.

7.3 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.

7.3.1 Snoops resulting from 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 [13, 4, 14, 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 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 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 CMP are sent to support cross-modified code (XMC).

7.3.2 Snoop flows

Figure 42 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 (sL2). The L2 is accessed on any L1 instruction or L1 data cache miss by all four cores, 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 L2 cache are queued in a common hardware structure called the *L2 queue*, and L2 access is pipelined. The internal and external snoop requests are queued in another hardware

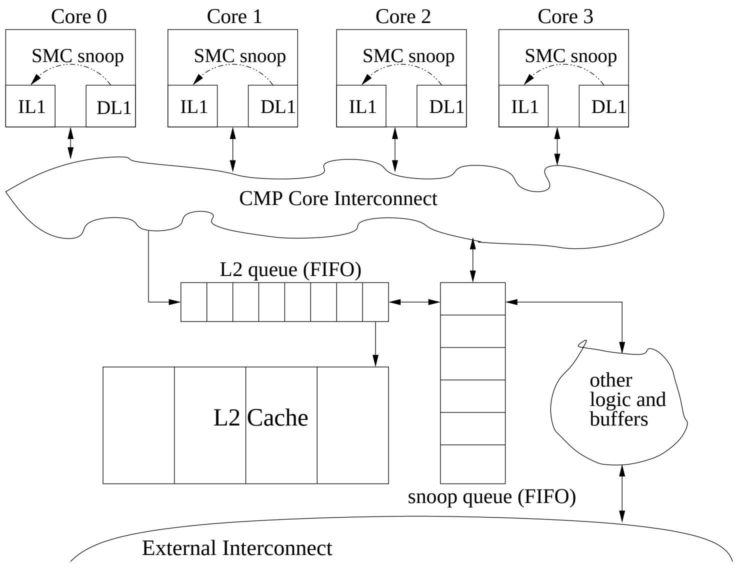


Figure 42: Snoop flows in a quad-core CMP system.

structure called the *snoop queue* [7, 48]. 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, and MSHRs (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 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

Table 13: Snoop triggers and snooped units in a quad-core system.

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	snoop
Code Fetch	Event Trigger	SMC snoop	snoop store buffer only	snoop	-	XMC snoop	snoop store buffer only	snoop	SMC snoop
Shared L2 evict	-	snoop	-	snoop	-	snoop	-	snoop	snoop
On store address dispatch	SMC snoop to iL1	-	-	-	-	-	-	-	-

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.

7.3.3 Snoop triggers and snoop probes

Table 13 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, 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 RFOs (request-for-ownership) 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 that reaches the last level cache is much more expensive than a data read or RFO (request-for-ownership) 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

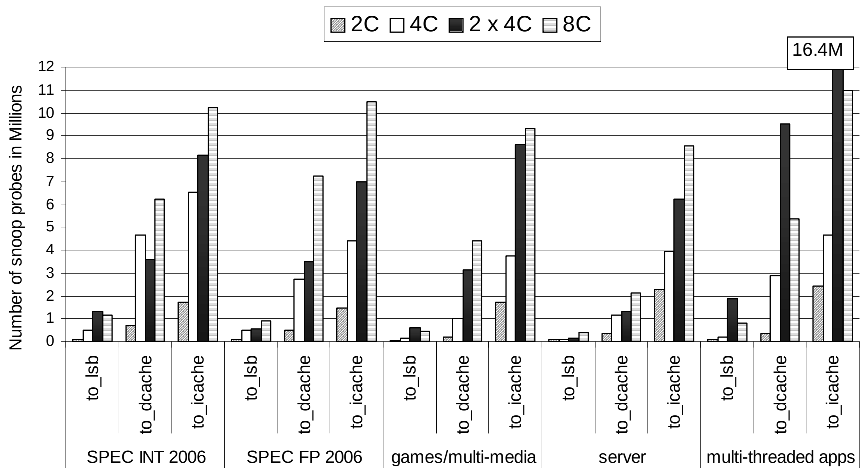


Figure 43: Snoop probes in different benchmarks.

number of cores per die and vertical cache hierarchy increases, these internal snoops will likely bottleneck the performance and power consumption.

Figure 43 and Figure 44 show the number of snoop probes received by each hardware structure for various benchmark categories in different processor configurations. The graph shows the results collected from around 200 traces for 6M instructions each based on the simulation methodology described in the experimental section. Each core in all configurations has split L1 instruction and data caches. All cores in one processor share the same L2. Also note that L2 employs a weakly-inclusive policy, where a cache line in L1 may not be in L2. Figure 43 shows the number of snoops to three microarchitecture modules, namely, Load-Store Buffer (LSB), L1 data cache, and 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 LSB.

Figure 44 shows the aggregate number of snoops for six different configurations. 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

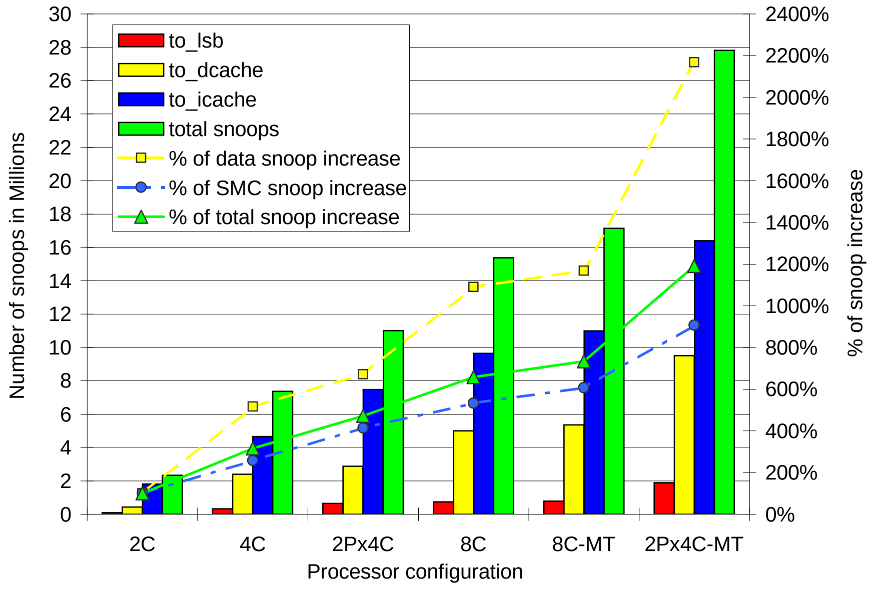


Figure 44: Snoop probes and snoop rate in different processor configuration.

has four cores. Each core is running one copy of a single-thread application. The 2Px4C-MT contains two quad-core processors running an eight-way multi-threaded application while 8C-MT is simply an eight-core system running the same multi-threaded application. Figure 44 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 snoop traffic as a result of external snoops in a dual-processor configuration 2Px4C compared to the uni-processor configuration 4C. And 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 the shared variables and external snoops. The secondary axis of Figure 44 shows the percentage snoop increase with respect to the 2C configuration as the baseline. Although both Figure 43 and Figure 44 show that the number of snoops to 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 44. As the number of cores increases beyond eight or 16 cores, snoop probes to the data cache and Load-Store Buffer (LSB) structures will limit performance. The number of snoops tends to increase in multi-threaded applications and is further aggravated in an 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 and the nature of the application is not conveyed by the program to the processor. To address these issues, 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.

7.4 Selective Snoop Probe (SSP)

In this section, we describe and discuss our hardware solution, which selectively filters snoop probes for requests reaching the last level cache. Each access to the last level cache spawns snoop probes, as described in Table 13, to obtain the most up-to-date copy of 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 that snoop probes generated as a result of stack access requests should always receive a clean response from the other cores as stack memory region is considered private to its own thread running on a core, as described in Section 7.2. 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 region and those that receive a clean response by 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 that respond, in fact, is much less than the number of snoop probes sent. The two main reasons for receiving clean responses are that: 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 distinguish between these types of variables, resulting in 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 proposed a simple hardware technique called *Selective Snoop Probe* (SSP). It uses 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 [76]. We profiled several benchmark suites to determine the distribution of memory accesses that reach the last level cache. Figure 45 shows the results of stack and non-stack access distribution. On average, about 30% of memory references go to stack using stack and frame pointers as source or destination registers. As shown in [77], the dominant method to access stack memory is via a stack pointer register and/or frame pointer register in an IA32 architecture. Other means such as from a general purpose register are not completely prohibited but are rather uncommon. By decoding the source or destination register identifier, a processor can isolate memory instructions that go to the stack region. 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 last level cache carries this annotation as part of the operation.

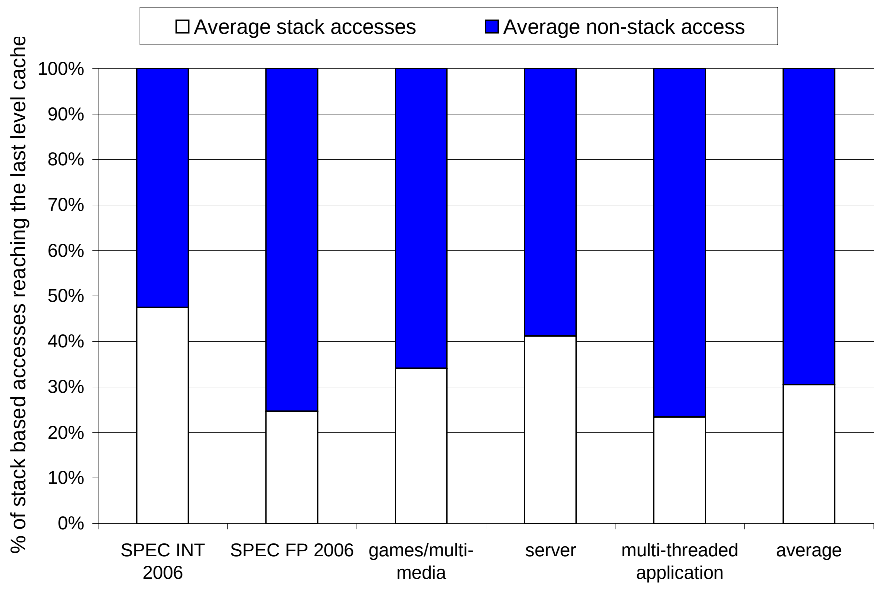


Figure 45: Stack accesses to the last level cache.

Figure 46 shows the details of internal and external snoop filtering mechanisms based on the program 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 [37] have been widely used in various microarchitecture optimization techniques for optimizing performance and power [85, 97, 107, 56, 93]. 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, two distinct counting Bloom filters labeled ① and ② in Figure 46 were added to each core, one to track valid cache lines in the instruction cache and eliminate unnecessary SMC snoops, and the other to track data cache lines and eliminate those 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 46. 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.

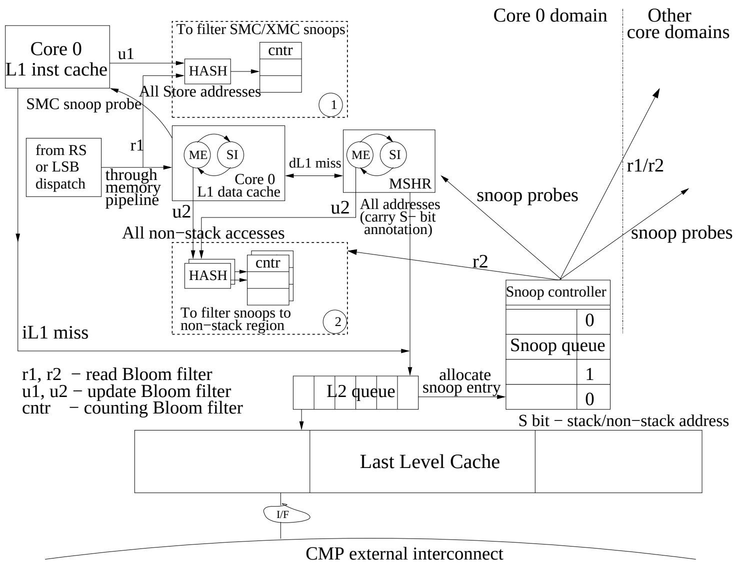


Figure 46: Selective Snoop Probe (SSP).

7.4.1 Selective snoop probe for the SMC (SSP-SMC)

The counting Bloom filter inside the box labeled ① in Figure 46 tracks all valid lines in the instruction cache. Whenever an instruction cache line becomes invalid, it is removed from this Bloom filter. The addition and removal from 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 RS (Reservation Station) or LSB (Load-Store Buffer). 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 instruction cache are eliminated.

7.4.2 Selective snoop probe for the stack region (SSP-SR)

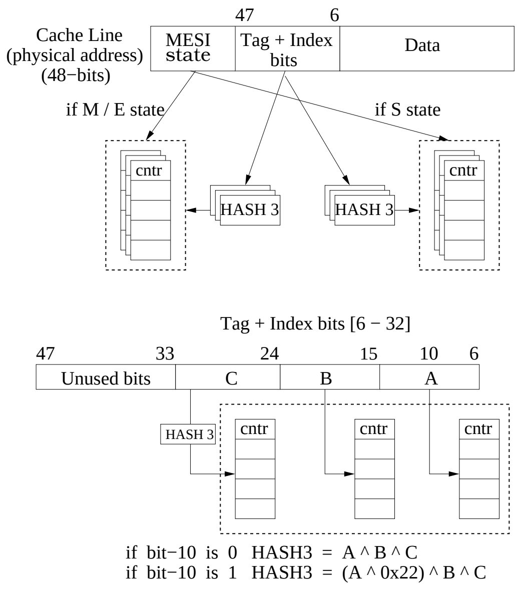
The requests that reach the last level cache carry stack-bit (S-bit) annotation as part of the operation. As described earlier, the S-bit annotation is set in the decode stage based on the source and destination register identifiers. The snoop controller looks at the S-bit before spawning snoop probes to determine if it is necessary to do so. If the S-bit is set, snoop controller do not send snoop probes and are thus eliminated. 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 last level cache accesses, as shown in Figure 45, for which snoop probes are eliminated by looking up the S-bit.

7.4.3 Selective snoop probe for the non-stack region (SSP-NSR)

On average 70% of requests reaching the last level cache go to the non-stack region. The counting Bloom filter inside the box labeled ② in Figure 46 tracks all non-stack accesses that look up data cache and MSHR as denoted by lines marked u2. The snoop controller looks up the counting Bloom filters in all the other cores based on Table 13 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 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 because of 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 our SSP technique are discussed in the following sections.

7.4.4 Bloom filter and hash function

The counting Bloom filter inside box ② in Figure 46 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 47.



Cache Line (physical address) (48−bits)
47
6
MESI state
Tag + Index bits
Data
if M / E state
if S state
cntr
HASH 3
HASH 3
cntr
Tag + Index bits [6 − 32]
47
33
24
15
10
6
Unused bits
C
B
A
HASH 3
cntr
cntr
cntr
if bit−10 is 0 HASH3 = A ^ B ^ C
if bit−10 is 1 HASH3 = (A ^ 0x22) ^ B ^ C

Figure 47: Hash functions used in counting Bloom filters.

The reason for segregating Bloom filters is to reduce aliases and thereby 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 47. Note that the hash functions are fixed. We did not use different hashes for different benchmark programs based on profiling in our experiments. 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 47. This combination of hashing is done to distribute the indexing of addresses to all entries and reduce aliases. The Bloom filter has 9-bit counters, as there are 64 sets and eight ways in the 32KB cache. The 9-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 and 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 L1 will become invalid after eager writeback.

The number of snoop probes eliminated by SSP is sensitive to the size of the Bloom filter array. The false-positive rate decreased as array size increased. 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, which alleviate the false-positives.

In summary, there are several advantages of using SSP. 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.

7.5 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 that: 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 the 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 in previous sections 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 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 performance may be better off if core affinity [19, 102] 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 for obtaining the potential modified lines when the OS migrates the thread during 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 last level cache just after the context switch. Normally, the context switch can be identified when important control registers [12] 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 in order to save power [20, 2]. 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.

7.6 Essential Snoop Probe (ESP)

The Selective Snoop Probe (SSP) technique use 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. The non-shared global and heap variables are identified by the counting Bloom filters. 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). The ESP technique is a simple and complexity-effective mechanism that exploits the non-shared information in all types (stack, global, and heap) of the variables used in the program to reduce snoop probes down to the essential ones.

The Essential Snoop Probe (ESP) technique requires synergy between compiler and hardware to work effectively. The compiler, through various techniques explained further below, annotates all 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 48 shows the implementation of the ESP technique. As shown in Figure 48, when the SMN bit is set, a load access 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*).

To address the self-modifying code condition, compiler needs to pass information if the program contained self-modified code to microarchitecture, identified either

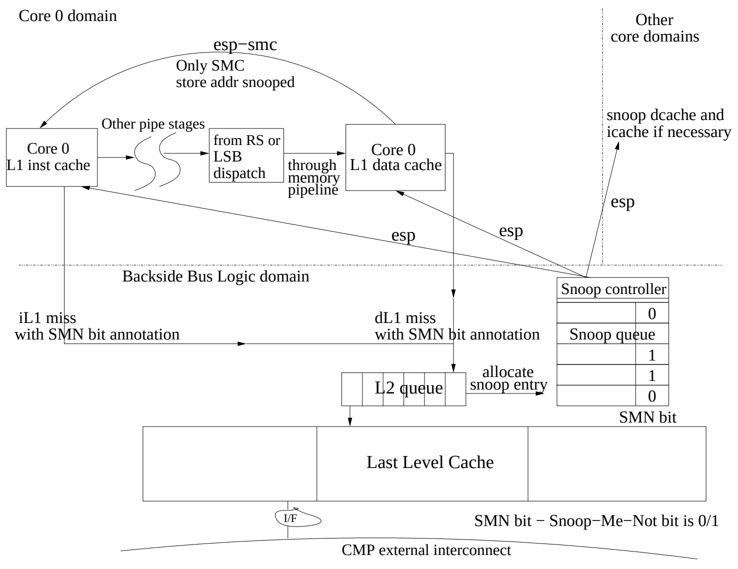


Figure 48: Essential Snoop Probe (ESP).

from the data flow analysis or from the user defined *pragmas*. One approach is to use the compiler tool chain assign a non-zero value to a predefined memory location. The predefined memory location can be in one of the sections that is part of the executable to indicate if the program contained self-modified code. The loader while loading the program reads this predefined memory location and set the SMC control register (SMC-CR) bit during the program initialization phase before program starts its execution. The microarchitecture checks this control register (SMC-CR) bit and prevents sending snoop probes to instruction cache when it is set. When the SMC-CR bit is set to indicate that it is indeed executing a self-modified code, the microarchitecture sends snoop probes to the instruction cache. While executing self-modifying code, the SMC snoop probes (line denoted by *esp-smc*) to instruction cache are sent. The lines *esp* and *esp-smc* indicate the essential snoop probes.

Compilers can use multitude of techniques to generate the SMN bit. Using the data flow analysis and algorithms [8], inter-procedure optimization, and other techniques, compilers can determine whether variables in a program are shared with other threads/programs. For example, variables are explicitly declared as shared, private, etc in the OMP [17] construct used in multi-threaded programming supported by many compilers [15, 87]. To support parallel programming, the POSIX thread interface provides a better way of dealing with thread local storage (TLS) [10] 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 the shared variable. 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 last level cache. To achieve this, the compiler needs to perform data flow analysis to determine 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 the legacy code or when running the code generated by a compiler that does not support SMN bits.

7.7 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 multi-core 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 [11, 92] or concentrated mesh [31] design will be efficient for future large-scale CMP systems. In the hierarchical cache coherence protocol design, the snoopy-based cache coherence protocol, which 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, 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.

7.8 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 traces collected from the real-world applications including the external events such as DMA and interrupts. The traces are gathered for

Table 14: Benchmark programs.

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

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 categories. The applications in each category are shown in Table 14. Each category has ten applications, and each application has multiple traces that represent different characteristic portion of the application similar to the SimPoint [60] 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

Table 15: Processor model parameters.

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

hardware structures. The simulated configurations are two-core, four-core, eight-core, and 2x4-core (two processor, four-cores per processor). A total of 500 traces were executed and the simulated processor configuration is shown in Table 15. We use the CACTI 4.2 [6] 70nm model to determine the energy consumed by the instruction cache tag, the data cache, and the hash arrays to evaluate the energy savings. The energy consumed by the 32KB cache is 0.4673 nJ, 2.466nJ when all banks are read, and the three 512-entry nine-bit Bloom filter is 0.0096 nJ. The leakage energy for 32KB cache is 0.1037 nJ, around 22.3% of the active energy.

Figure 49 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 the die, as the number of snoops to all the cores increases. The data cache energy savings in the 2Px4C dual-processor configuration is little bit higher than the 4C

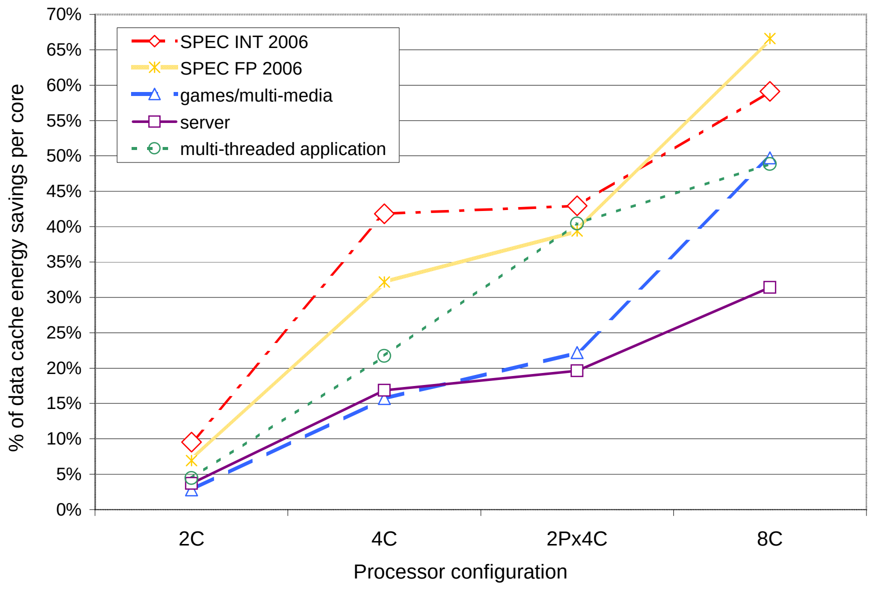


Figure 49: Energy savings in data cache per core using SSP.

configuration, because of the external snoop (snoop probes between CMP processors) filtering in the 2P case.

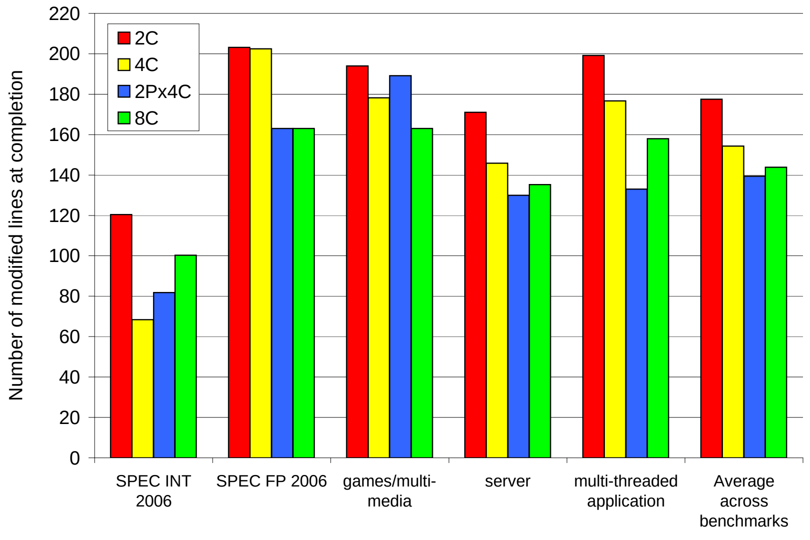


Figure 50: Number of modified lines after the program completion.

Figure 50 shows the number of modified lines after the simulation completed. We also consider the number of cycles required to flush these modified lines to the last level cache to support the eager writeback.

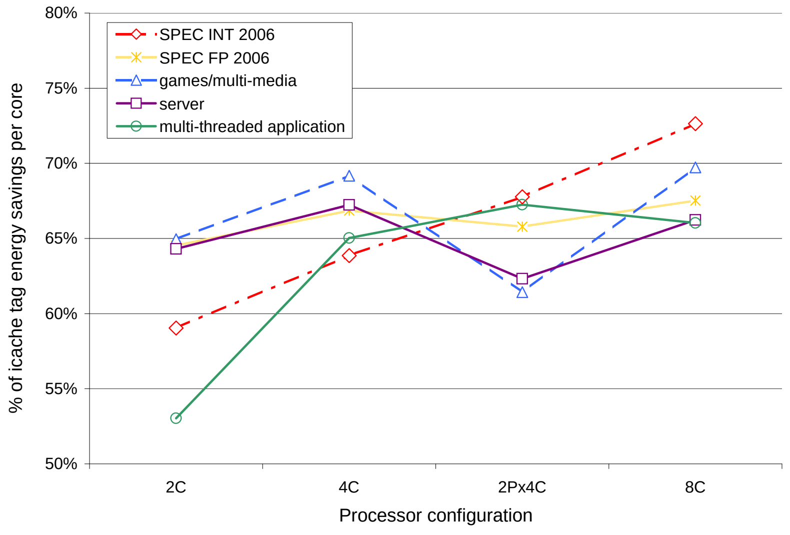


Figure 51: Energy savings in the instruction cache tag per core using SSP.

Figure 51 shows the percentage of instruction cache tag energy savings in snoop probes using the SSP technique. It shows that 50% to 70% of snoop probe 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 (Request For Ownership). 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 do not vary much, the percentage of energy savings in the instruction cache tag also do not vary widely.

Figure 52 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 caches provides more opportunity to service the regular loads and stores that are needed by the core to make progress.

We implemented the RegionScout [84] technique proposed earlier by Moshovos *et al.* to compare with the SSP technique. We use a 64-entry fully-associative NSRT

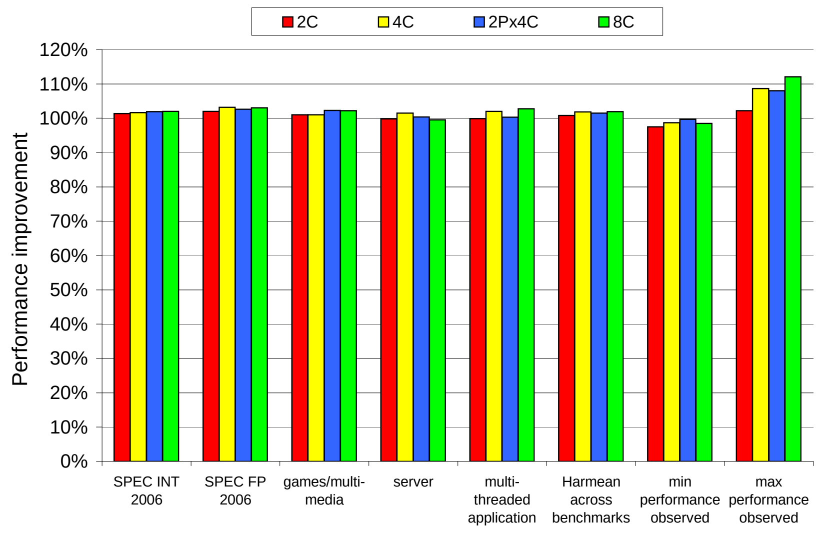


Figure 52: Performance impact using SSP.

(Not Shared Region Table) cache, and a 512-entry CRH (Cached Region Hash) to record the regions that are locally cached with an 8K region size. On average the SSP technique sends only 30-35% of snoop probes sent by RegionScout. Figure 53 compares the energy savings achieved using the SSP and RegionScout techniques.

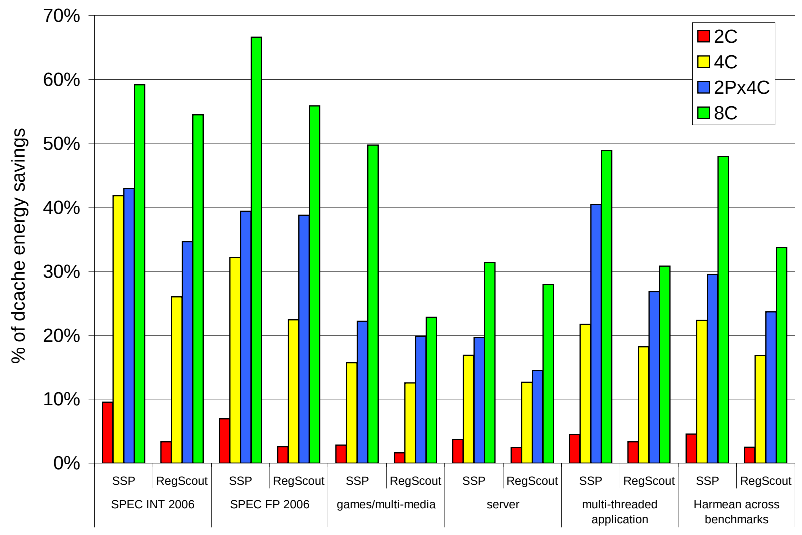


Figure 53: Energy savings comparison in data cache using SSP and RegionScout.

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 compilers can implement a variety of techniques to detect the sharing of the variables used in the program. Figure 54 shows the percentage of cache energy that is 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 eight-core configuration) is 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 54 shows that on average 85% of SMC snoop probes to the instruction cache tag is non-essential. This percentage of savings does not vary much across the benchmark categories and the processor configurations. The ESP technique uses the synergy between the processor and the compiler to achieve higher energy savings compared to the SSP technique.

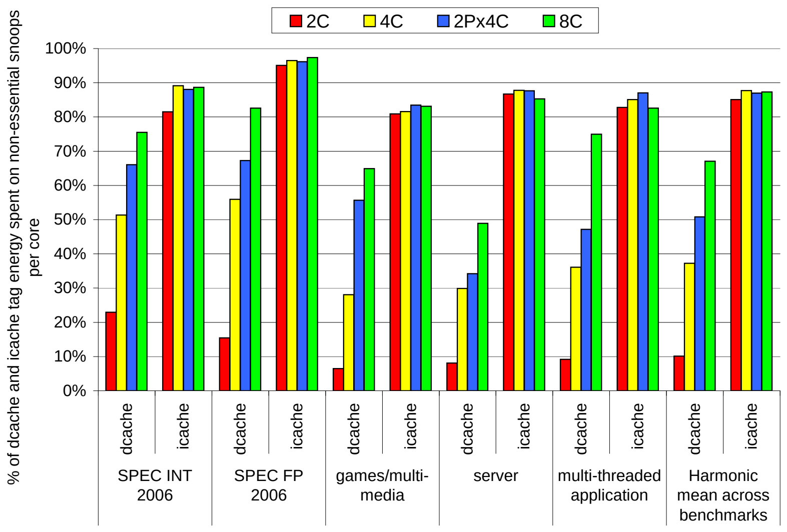


Figure 54: Energy savings in data cache and instruction cache per core using ESP.

7.9 Summary

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

The advantage of using the SSP technique is that, all the functionality is implemented in the hardware and transparent to the programmers. In addition, previously compiled binaries can benefit from this technique without the need for 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 was spent on the non-essential snoop probes that can potentially be saved using the compiler guided ESP technique.

CHAPTER VIII

RELATED WORK

8.1 Low-power caches and TLBs

Many memory partitioning schemes at the architectural level were proposed for energy reduction. These techniques can be classified into two categories: horizontal partitioning and vertical partitioning. The vertical partitioning techniques, such as line buffers [55, 103] and filter caches [73], attempt to capture cache lines in a small intermediate structure to exploit the transient data locality at the cost of performance loss. The sub-banking [55], selective cache ways [24], and the PSAC [62] partitioned the caches into horizontal segments and enable only the partitions that are being accessed at runtime. The HP3000 Series II [36] featured a stack cache with a FIFO replacement policy as an extension to the main memory in the absence of a data cache. The CRISP processor [35] performed register allocation on the 32-entry stack cache at runtime to avoid the overheads of procedure calls. Cho et al. [44] proposed a decoupled stack cache to alleviate the performance degradation in a multi-ported cache. Lee and Tyson proposed region-based caching [78], a preliminary concept for semantic-aware data caches. Collins *et al.* [46] studied a pointer cache to enhance indirect loads. Their mechanism recognizes heap address accesses dynamically based on a method in [47] and inserts them into the pointer cache.

Ashok *et al.* [25] used a compiler-managed address speculation to enable tagless cache accesses for power reduction. Kadayif et al. [69] proposed a mechanism to reduce the power in the i-TLB by adding a register called Current Frame Register (CFR) for address translation. Instead of looking up the i-TLB for each instruction, the processor fetches the translated address from the CFR unless there is a memory page change. Our intra-cycle compaction differs from this, as we also exploited redundancy among synonymous accesses in the same cycle, while the scheme in [69] only eliminates lookups in subsequent cycles after the CFR has been filled. A compiler-based data layout combined with a small translation register was proposed by Kandemir *et al.* [70] to reduce the d-TLB energy. The scheme is similar to our inter-cycle compaction, but requires re-compilation. The way-prediction [65] technique was proposed to reduce cache energy by speculating one predicted way instead of looking up all the ways in a set-associative cache. Other approaches for the TLB power reduction include selective filter-bank TLB [79].

Austin and Sohi [28] proposed re-translation and piggybacking ports to increase the address translation bandwidth. To achieve higher bandwidth, a four-ported eight-entry pre-translation cache was added in the ID stage. The output of the pre-translation cache is attached to the TLB entry in the ID stage, making the TLB translation available at the start of the execution speculatively. The first difference is, piggybacking the translation happens for the next cycle, unlike our intra-cycle compaction technique that happens for the current cycle. The second difference is, the high bandwidth is achieved at the expense of higher power and inter-cluster changes to the baseline architecture. The mechanisms that we proposed consume less power and require smaller hardware changes only in the memory cluster.

Hammerstrom and Davidson [61] proposed a method to estimate the information content of memory references and the address generation overhead. They proposed a few optimization techniques to reduce the address calculation overhead based on the measured entropy. Becker *et al.* [34] proposed a scheme to dynamically encode a typical 32-bit address into four to seven bits using Huffman coding. Their experiments show that 83% of the address bits in the traces contain redundant information.

A software-controlled virtual page number mechanism was used by Petrov and Orailoglu [89] to reduce the TLB energy. Their method consists of two major phases, off-line and on-line phases. In the off-line phase, a special algorithm code is inserted prior to entering and after exiting the hot-spot with reduced tag information. Additionally, the compiler also attached special tables with information about the hot-spots. The OS would provide the information regarding the libraries during the on-line phase. Our techniques require only the hardware change and do not need any profile feedback support.

Juan *et al.* [68] proposed a circuit design to modify the CAM cell by adding a transistor in the discharge path. With the modified cell, the control line can be used to pre-charge the match line without resetting the bit lines. The bit lines change because of the changes in the virtual addresses of consecutive accesses. As the unified TLB is accessed by all the memory references from different semantic regions, the entropy is higher. This leads to a change of four bits per access. This is different from our semantic-aware technique, where the addresses are segregated based on different memory regions.

A number of TLB power reduction schemes have been proposed for the regular C/C++ applications. Kadayif *et al.* [69] added a register called Current Frame Register (CFR) to the instruction address translation. Instead of looking up the i-TLB, the processor fetches the translated address from the CFR unless there is a memory page change. Way-prediction [65] was proposed to reduce cache energy by speculating one predicted way instead of looking up all ways in a set-associative cache. Other approaches for TLB power reduction include selective filter-bank TLB [79] and semantic-aware memory [76].

Ramesh *et al.* [90] studied the characteristics of Java applications and SPECjvm98 suite and analyzed the memory, object size footprint, and instruction mix of the Java applications running on JVM. There interaction between the Java application and the JVM on which it is running was studied by Georges and Tia [54, 86]. It has been shown [101] that the dTLB miss rate of SPECjvm98 benchmark is around 2%

that is much higher than 0.1% reported for SPEC95 benchmarks written in C/C++ workloads [74]. Similarly, the L1 data cache miss rates and more specifically the data write miss rates are also higher [91, 54].

Kim *et al.* [71] analyzed the memory reference patterns of applications in the SPECjvm98 suite and found that the size and the lifetime of the objects is small. Vijaykrishnan *et al.* [106] proposed an architectural support for object manipulation, stack processing, and two-level hybrid cache to improve the performance of Java applications. Shimizu and Kon [100] extended the basic idea in traja processor of java object lookaside buffer for Java applications running in interpreted mode or directly on Java processor. Java object lookaside buffer speeds up the resolution of constant pool references.

Vijaykrishnan *et al.* [105] presented a characterization of the energy consumption by the caches and the main memory when executing the SPECjvm98 benchmarks in the JIT and interpreter modes. The energy consumption is profiled for different hardware and software configurations. They made the observation that from the energy perspective JIT mode is better than the interpreter mode. The main memory energy consumption is more dominant than that of the caches, with the main contributor being data references. The energy consumed in the JIT mode is mainly due to code installation and the subsequent misses. Chen *et al.* [42] modified the cache organization to include a small cache for storing the dynamically generated native code to reduce the energy consumption in the caches for the Java applications running on JVM. Kim *et al.* [72] proposed an energy efficient Java execution using local memory and object co-location. In the object co-location, they study the static profile of Java programs running on JVM and use the results to intelligently place the objects in the heap space such that it reduces the energy consumption.

8.2 Prior techniques for snoop power reduction

The earlier research work has shown that many snoop requests miss in all the remote caches/nodes in a shared memory system [85, 52, 94, 40, 26]. But earlier work has not delved into the reason for these large misses in the other nodes. As indicated in this dissertation, the two main reasons for these large misses are: 1) the stack accesses do not have global visibility, and 2) many programs do not share variables with other programs/threads that are running in the system.

None of the earlier proposed hardware or software solutions were cognizant of the semantics of the variables used in the program. There is a key difference between our ESP technique and the techniques proposed by others. The ESP technique 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 (eg., pages) and locality to optimize the snoops. The Page Sharing Table technique proposed by Ekaman [52] *et al.* used vectors to identify the sharing at page level. Saldanha *et al.*, proposed serial snooping [94] to reduce the energy in shared multiprocessors. In the Include Jetty [85] 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 [50], they do not work as effectively when applied to CMP. The reasons are that: 1) the include Jetty does not prevent a majority of the unnecessary snoops, and 2) the Exclude Jetty requires prohibitively large hardware to work effectively. Our SSP and ESP techniques work well in the CMP environment, as they selectively send snoop probes where needed.

Recently, Moshovos proposed RegionScout [84], where each node can determine in advance if a request would miss in all the 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 knows that it does not need to probe any other node. One key difference is that the SSP and ESP techniques selectively send the snoop probes or completely eliminate them if possible, whereas the RegionScout has to broadcast the initial request to identify the Region Hit information. The RegionScout technique requires a bus interconnect architecture with a wired-OR signal to notify the region hit. In contrast to RegionScout, our techniques are not limited by the choice of the interconnect architecture used in the CMP systems.

Cantin *et al.* [40] proposed a technique to reduce the number of broadcast snoops required to maintain the coherency in an SMP system. Their idea is similar to the RegionScout and requires a hardware structure called Region Coherency Array as well as extra bits in the processor interconnect. The SSP technique presented is different because it completely eliminates the snoops for stack accesses and selectively dispatches snoops for non-stack accesses. In other words, the Region Coherency Array divided the memory accesses into those requiring broadcast to all the nodes and those that do not, whereas we generalize this division to the one that exactly needs snoop probes. The ESP technique is completely different from the RegionScout and the Region Coherency Array, as it uses the compiler support to reduce snoop probes to the essential ones. It uses the semantics and sharing properties of the program variables without any storage-based hardware structures.

Atoofian and Banisadi [26] proposed a power-saving technique based on the observation that the snoop responses exhibit high locality. Their technique maintains saturating hardware counters to predict the likely node in an SMP system that will provide a hit or hit-modified response to a snoop probe. Using this predictor, a snoop probe is sent out to the predicted node first. Then the snoop is broadcasted only if it gets a clean response. Although, this technique reduced snoops by selectively sending them out to the predicted nodes, they do not completely eliminate the snoops for the non-shared variables that constitute a major portion of the snoops.

CHAPTER IX

CONCLUSIONS

The semantics of memory references and the application behavior provides vital information that can be used to design a power and performance efficient processor. We took a holistic view of the power consumption problem in the memory sub-system and proposed a semantics-oriented framework that exploits the fundamental meaning of memory references in the applications to reduce processor power.

We observed the following important memory references characteristics:

- A considerable portion of memory references (around 40%) are to the stack region and have very good locality.
- We found that the concurrent and consecutive memory accesses are often synonymous, going to the same page.
- The entropy of the stack virtual page numbers is low resulting from the good spatial and temporal locality of stack references. The entropy of the higher-order bits of global virtual page numbers is low, as the size of the global data is determined and fixed during the program compilation.
- The instruction TLB and the data TLB are accessed by the JVM code, its associated data, and also by the Java application code and data. The interference between these accesses affect the power and performance of the Java applications.
- The snoop probes increase with the number of cores on the die. The main reason that many of the snoop probes get clean response is that the knowledge about the semantics of the memory references, shared variables in the program, and the nature of the application is not conveyed by the program to the processor. Also, the existing snooping cache coherence protocol and its conservative implementation

indiscriminately broadcast the snoop probes to all the cores.

Based on these observations and analysis, the thesis made five different contributions and demonstrated that the semantics-oriented framework can be used to reduce the power consumed by the TLBs and caches. The first four techniques concentrated on reducing the power consumed by the TLBs using the semantics and characteristics of the memory references. The fifth one deals with the power consumed by the snoop probes in the CMP systems using the semantics and properties of the variables used in the application.

The hardware/software technique that uses compiler can be extended to heterogeneous systems to reduce the snoop probes to the instruction and the data cache. In a heterogeneous system, like AMD Fusion, the regular program and the graphics program run on their respective cores. In this kind of a system, we expect the number of snoops initiated from the graphics cores to the other multiple cores on the die to be much higher based on the nature of the graphics workloads. We expect our proposed hardware supported sofware based technique if used by the 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.

We believe that the semantics-oriented framework can be extended to other parts of the microarchitectural units to design a low-power and performance efficient microprocessor.

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] "Blackford: A dual processor chipset for servers and workstations," in *www.hotchips.org/archives/hc18/3_Tues/HC18.S9/HC18.S9T3.pdf*.

[6] "CACTI 4.2," in *http://quid.hpl.hp.com:9081/cacti*.

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

[8] "Codesurfer," in *www.grammatech.com/products/codesurfer/overview.html*.

[9] "Dynamic SimpleScalar 1.0.1." http://www-ali.cs.umass.edu/DSS/index.html.

[10] "ELF handling for Thread Local Storage," in *people.redhat.com/drepper/tls.pdf*.

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

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

[13] "Intel Architecture Optimization Reference Manual," in *http://developer.intel.com/design/PentiumII/manuals/245127.htm*.

[14] "Intel Itanium Architecture," in *www.intel.com/design/itanium/manuals/ iiasdmanual.htm*.

[15] "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*.

[16] "JikesRVM Home Page." http://jikesrvm.sourceforge.net/.

[17] "OpenMP application program interface," in *www.openmp.org/drupal/mp-documents/spec25.pdf*.

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

[19] "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*.

[20] "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*.

[21] "Power5 Architecture," in *www-941.ibm.com/collaboration/wiki/ display/LinuxP/POWER5+Architecture*.

[22] "Spec JVM 98 Benchmarks." http://www.spec.org/osg/jvm98.

[23] "The MIPS 64 architecture." http://www.mips.com/content/Products/ Architecture/MIPS64.

[24] Albonesi, D. H., ""Selective Cache Ways: On-Demand Cache Resource Allocation"," in *International Symposium on Microarchitecture*, 1999.

[25] Ashok, R., Chheda, S., and Moritz, C. A., "Cool-Mem: combining statically speculative memory accessing with selective address translation for energy efficiency," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, 2002.

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

[27] Austin, T., Larson, E., and Ernst, D., "Simplescalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, 2002.

[28] Austin, T. M. and Sohi, G. S., "High-bandwidth Address Translation for Multiple-issue Processors," in *Proceedings International Symp. on Computer Architecture*, 1996.

[29] Baer, J.-L. and Wang, W.-H., "On the inclusion properties for multi-level cache hierarchies," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1988.

[30] Bailey, A., "Barcelona's Innovative Architecture is Driven by a New Shared Cache." http://developer.amd.com/articles.jsp?id=173&num=1.

[31] Balfour, J. and Dally, W. J., "Design tradeoffs for tiled cmp on-chip networks," in *Proceedings of the 20th annual international conference on Supercomputing*, 2006.

[32] Ballapuram, C. S., Lee, H.-H. S., and Prvulovic, M., "Synonymous Address Compaction for Energy Reduction in Data TLB," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2005.

[33] Barroso, L. A., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R., and Verghese, B., "Piranha: A Scalable Architecture Based on Single Chip Multiprocessing," in *Proceedings of the International Symposium on Computer Architecture*, 2000.

[34] Becker, J. C., Park, A., and Farrens, M., "An analysis of the information content of address reference streams," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, 1991.

[35] Berenbaum, A. D., Ditzel, D. R., and McLellan, H. R., "An Introduction to the CRISP Architecture," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, 1987.

[36] Blake, R. P., "Exploring a Stack Architecture," *IEEE Computer*, May 1977.

[37] Bloom, B. H., "Space/time Trade-offs in Hash Coding with Allowable Errors," *Communication of the ACM*, vol. 13, no. 7, 1970.

[38] Brooks, D., Tiwari, V., and Martonosi, M., "Wattch: a Framework for Architectural-level Power Analysis and Optimizations," in *Proceedings of the International Symposium on Computer Architecture*, 2000.

[39] Brooks, D., Tiwari, V., and Martonosi, M., "Wattch: a Framework for Architectural-level Power Analysis and Optimizations," in *Proceedings of International Symposium on Computer Architecture*, 2000.

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

[41] Cao, Y., Sato, T., Sylvester, D., Orshansky, M., and Hu, C., "New paradigm of predictive mosfet and interconnect modeling for early circuit design," in *Proceedings of the IEEE Custom Integrated Circuit Conference*, 2000.

[42] Chen, G., Kandemir, M., Vijaykrishnan, N., and Irwin, M., "Energy-aware code cache management for memory-constrained java devices," in *Proceeding of International conference on systems-on-chip*, pp. 179–182, 2003.

[43] Chen, J. B., Borg, A., and Jouppi, N. P., "A simulation based study of TLB performance," in *Proceedings of the 19th annual International Symposium on Computer Architecture*, 1992.

[44] Cho, S., Yew, P.-C., and Lee, G., "Decoupling local variable accesses in a wide-issue superscalar processor," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, 1999.

[45] Choi, J.-H., Lee, J.-H., Jeong, S.-W., Kim, S.-D., and Weems, C., "A Low power TLB structure for Embedded Systems," *IEEE Computer TCCA Letter*, January 2002.

[46] Collins, J., Sair, S., Calder, B., and Tullsen, D. M., "Pointer cache assisted prefetching," in *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, 2002.

[47] Cooksey, R., Jourdan, S., and Grunwald, D., "A stateless, content-directed data prefetching mechanism," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, 2002.

[48] Dahlen, E., Gustin, J., Meredith, S., and Moran, D., "The 82460GX Sever/Workstation Chip Set," *IEEE Micro*, vol. 20, no. 6, pp. 69–75, 2000.

[49] Dinan, J. and Moss, E., "DSSWattch: Power Estimation in Dynamic SimpleScalar," in *Technical Report, UMass ALI Lab, Amherst, MA*.

[50] Ekman, M., Dahlgren, F., and Stenstrom, P., "Evaluation of Snoop Energy-Reduction techniques for Chip-Multiprocessors," *Workshop on Duplicating, Deconstructing and Debunking in conjunction with the International Symposium on Computer Architecture*, 2002.

[51] Ekman, M., Dahlgren, F., and Stenstrom, P., "TLB and snoop energy-reduction using virtual caches in low-power chip-multiprocessors," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2002.

[52] Ekman, M., Stenstrom, P., and Dahlgren, F., "TLB and Snoop Energy-reduction using Virtual Caches in Low-power Chip Multiprocessors," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2002.

[53] Fan, D., Tang, Z., Huang, H., and Gao, G. R., "An energy efficient tlb design methodology," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 351–356, 2005.

[54] Georges, A., Buytaert, D., Eeckhout, L., and Bosschere, K. D., "Method-level phase behavior in java workloads," in *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 270–287, 2004.

[55] Ghose, K. and Kamble, M. B., "Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation," in *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, 1999.

[56] Ghosh, M., Ozer, E., Biles, S., and Lee, H.-H. S., "Efficient System-on-Chip Energy Measurement with a Segmented Bloom Filter," in *ARCS '06*, 2006.

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

[58] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B., "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *IEEE 4th Workshop on Workload Characterization*, 2001.

[59] Halfhill, T. R., "MIPS 74K Goes Superscalar," *Microprocessor Report*, Vol. 21, No. 5, May 2007.

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

[61] Hammerstrom, D. and Davidson, E. S., "Informaiton content of cpu memory referencing behavior," in *Proceedings of the International Symposium on Computer Architecture*, 1977.

[62] Huang, M., Renau, J., Yoo, S.-M., and Torrellas, J., "L1 data cache decomposition for energy efficiency," in *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, 2001.

[63] Huang, X., B.Moss, J. E., McKinley, K. S., Blackburn, S., and Burger, D., "Dynamic simplescalar: Simulating java virtual machines," in *Technical Report TR-03-03, University of Texas at Austin, Department of Computer Sciences*, 2003.

[64] Huffman, D. A., "A method for the construction of minimum redundancy codes," in *Proceedings of the IRE*, 1952.

[65] Inoue, K., Ishihara, T., and Murakami, K., "Way-predicting Set-associative Cache for High Performance and Low Energy Consumption," in *Proceedings International Symp. on Low Power Electronics and Design*, 1999.

[66] Jouppi, N., "CACTI 3.0." http://research.compaq.com/ wrl/people/ jouppi/CACTI.html, 1999.

[67] Juan, T., Lang, T., and Navarro, J. J., "Reducing TLB Power Requirements," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 1997.

[68] Juan, T., Lang, T., and Navarro, J. J., "Reducing tlb power requirements," in *Proceedings of the 1997 International Symposium on Low Power Electronics and Design*, 1997.

[69] Kadayif, I., Sivasubramaniam, A., Kandemir, M., Kandiraju, G., and Chen, G., "Generating physical addresses directly for saving instruction TLB energy," in *Proceedings International Symp. on Microarchitecture*, 2002.

[70] Kandemir, M., Kadayif, I., and Chen, G., "Compiler-Directed Code Restructuring for Reducing Data TLB energy," in *Proceedings International Conference on Hardware/Software Codesign and System Synthesis*, 2004.

[71] Kim, J.-S. and Hsu, Y., "Memory system behavior of java programs: methodology and analysis," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2000.

[72] Kim, S., Tomar, S., Vijaykrishnan, N., Kandemir, M., and Irwin, M., "Energy-efficient java execution using local memory and object co-location," in *IEE Proceedings in Computer and Digital Techniques*, vol. Vol.151 No.1, pp. 33–42, 2004.

[73] Kin, J., Gupta, M., and Mangione-Smith, W., "Filtering Memory References to Increase Energy Efficiency," *IEEE Transactions on Computers*, vol. Vol. 49, No. 1, 2000.

[74] Lee, D. C., Crowley, P. J., Baer, J.-L., Anderson, T. E., and Bershad, B. N., "Execution characteristics of desktop applications on windows nt," in *Proceedings of the International Symposium on Computer Architecture*, pp. 27–38, 1998.

[75] Lee, H.-H. S., Fryman, J. B., Diril, A. U., and Dhillon, Y. S., "The Elusive Metric for Low-Power Architecture Research," in *The Workshop on Complexity-Effective Design in conjunction with Internation Symposium on Computer Architecture*, 2003.

[76] Lee, H.-H. S. and Ballapuram, C. S., "Energy Efficient D-TLB and Data Cache using Semantic-aware Multilateral Partitioning," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2003.

[77] Lee, H.-H. S., Smelyanskiy, M., Newburn, C. J., and Tyson, G. S., "Stack Value File: Custom Microarchitecture for the Stack," in *Proceedings of International Symposium on High Performance Computer Architecture*, 2001.

[78] Lee, H.-H. S. and Tyson, G. S., "Region-based caching: an energy-delay efficient memory architecture for embedded processors," in *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, 2000.

[79] Lee, J.-H., Park, G.-H., Park, S.-B., and Kim, S.-D., "A Selective Filter-bank TLB System," in *Proceedings International Symp. on Low Power Electronics and Design*, 2003.

[80] Lin, R., Nakano, K., Olariu, S., and Zomaya, A., "An efficient parallel prefix sums architecture with domino logic," in *IEEE Transactions on Parallel and Distributed Systems*, 2003.

[81] Manne, S., Klauser, A., and Grunwald, D., "Pipeline gating: speculation control for energy reduction," in *Proceedings of the International Symposium on Computer Architecture*, 1998.

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

[83] Montanaro, J., Witek, R. T., Anne, K., Black, A. J., Cooper, E. M., Dobberpuhl, D. W., Donahue, P. M., Eno, J., Hoeppner, G. W., Kruckemyer, D., Lee, T. H., Lin, P. C. M., Madden, L., Murray, D., Pearce, M. H., Santhanam, S., Snyder, K. J., Stephany, R., and Thierauf, S. C., "A 160-mhz, 32-b, 0.5-w cmos risc microprocessor," *Digital Technical Journal*, vol. Vol.9 No.1, 1997.

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

[85] Moshovos, A., Memik, G., Falsafi, B., and Choudhary, A. N., "JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers," in *Proceedings of International Symposium on High Performance Computer Architecture*, 2001.

[86] Newhall, T., "Performance measurement of interpreted, Just-in-Time compiled, and dynamically compiled executions," in *Master's Thesis Report, University of Wisconsin, Madison, WI.*

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

[88] Papamarcos, M. S. and Patel, J. H., "A low-overhead coherence solution for multiprocessors with private cache memories," in *Proceedings of the International Symposium on Computer Architecture*, 1984.

[89] Petrov, P. and Orailoglu, A., “Virtual Page Tag Reduction for Low-power TLBs,” in *Proceedings of the International Conference on Computer Design*, 2003.

[90] Radhakrishnan, R., Rubio, J., and John, L., “Characterization of java applications at bytecode and ultra-sparc machine code levels,” in *Proceedings of the International Conference on Computer Design*, p. 281, 1999.

[91] Rajan, A., Hu, S., and Rubio, J., “Cache performance in java virtual machines: a study of constituent phases,” in *International workshop on workload characterization*, pp. 80–91, 2002.

[92] Ravindran, G. and Stumm, M., “A performance comparison of hierarchical ring- and mesh- connected multiprocessor networks,” in *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, 1997.

[93] Roth, A., “Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization,” in *Proceedings of the International Symposium on Computer Architecture*, 2005.

[94] Saldanha, C. and Lipasti, M., “Power efficient cache coherence,” *Workshop on Memory Performance Issuses in conjunction with Internation Symposium on Computer Architecture*, 2001.

[95] Sassone, P. G., Rupley, J., Brekelbaum, E., Loh, G. H., and Black, B., “Matrix Scheduler Reloaded,” in *Proceedings of the International Symposium on Computer Architecture*, 2007.

[96] Sery, G., Borkar, S., and De, V., “Life is CMOS: Why Chase Life After?,” 2002.

[97] Sethumadhavan, S., Roesner, F., Emer, J. S., Burger, D., and Keckler, S. W., “Late-binding: Enabling Unordered Load-store Queues,” in *Proceedings of the International Symposium on Computer Architecture*, 2007.

[98] Shannon, C. E., “Prediction and entropy of printed english,” in *Bell Systems Technical Journal*, vol. 30, 1951.

[99] Shi, W., Lee, H.-H. S., Falk, L., and Ghosh, M., “An Integrated Framework for Dependable and Revivable Architectures Using Multicore Processors,” in *Proceedings of the International Symposium on Computer Architecture*, 2006.

[100] Shimizu, N. and Kon, C., “Java object look aside buffer for embedded applications,” in *Proceedings of the 2003 workshop on Memory performance*, pp. 43–49, 2003.

[101] Shuf, Y., Serrano, M. J., Gupta, M., and Singh, J. P., “Characterizing the memory behavior of java workloads: a structured view and opportunities for optimizations,” in *Proceedings of SIGMETRICS international conference on Measurement and modeling of computer systems*, pp. 194–205, 2001.

[102] Squillante, M. S. and Lazowska, E. D., "Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, 1993.

[103] Su, C.-L. and Despain, A. M., "Cache design trade-offs for power and performance optimization: a case study," in *Proceedings of the 1995 International Symposium on Low Power Design*, 1995.

[104] Sundaramoorthy, K., Purser, Z., and Rotenberg, E., "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.

[105] Vijaykrishnan, N., Kandemir, M., Kim, S., Tomar, S., Sivasubramaniam, A., and Irwin, M. J., "Energy behavior of java applications from the memory perspective," in *In USENIX Java Virtual Machine Research and Technology Symposium (JVM'01), Monterey, CA, USA*, 2001.

[106] Vijaykrishnan, N., Ranganathan, N., and Gadekarla, R., "Object-oriented architectural support for a java processor," in *Proceedings of the European Conference on Object-Oriented Programming*, pp. 330–354, 1998.

[107] Woo, D. H., Ghosh, M., Ozer, E., Biles, S., and Lee, H.-H. S., "Reducing Energy of Virtual Cache Synonym Lookup Using Bloom Filters," in *CASES '06*, 2006.

[108] Ziv, J. and Lempel, A., "A Universal Algorithm for Sequential Data Compression," in *IEEE Transactions on Information Theory*, vol. IT-23, No. 3, 1977.