ARCHITECTURAL SUPPORT FOR HIGH-PERFORMANCE, POWER-EFFICIENT AND SECURE MULTIPROCESSOR SYSTEMS

A Dissertation

by

BAIK SONG AN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

August 2012

Major Subject: Computer Science

ARCHITECTURAL SUPPORT FOR HIGH-PERFORMANCE, POWER-EFFICIENT AND SECURE MULTIPROCESSOR SYSTEMS

A Dissertation

by

BAIK SONG AN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Eun Jung Kim
Committee Members,	James Caverlee
	Natarajan Gautam
	Guofei Gu
Head of Department,	Duncan M. Walker

August 2012

Major Subject: Computer Science

ABSTRACT

Architectural Support for High-Performance, Power-Efficient and Secure Multiprocessor Systems. (August 2012)

Baik Song An, B.S., Seoul National University; M.S., Seoul National University

Chair of Advisory Committee: Dr. Eun Jung Kim

High performance systems have been widely adopted in many fields and the demand for better performance is constantly increasing. And the need of powerful yet flexible systems is also increasing to meet varying application requirements from diverse domains. Also, power efficiency in high performance computing has been one of the major issues to be resolved. The power density of core components becomes significantly higher, and the fraction of power supply in total management cost is dominant. Providing dependability is also a main concern in large-scale systems since more hardware resources can be abused by attackers. Therefore, designing high-performance, power-efficient and secure systems is crucial to provide adequate performance as well as reliability to users.

Adhering to using traditional design methodologies for large-scale computing systems has a limit to meet the demand under restricted resource budgets. Interconnecting a large number of uniprocessor chips to build parallel processing systems is not an efficient solution in terms of performance and power. Chip multiprocessor

(CMP) integrates multiple processing cores and caches on a chip and is thought of as a good alternative to previous design trends.

In this dissertation, we deal with various design issues of high performance multiprocessor systems based on CMP to achieve both performance and power efficiency while maintaining security. First, we propose a fast and secure off-chip interconnects through minimizing network overheads and providing an efficient security mechanism. Second, we propose architectural support for fast and efficient memory protection in CMP systems, making the best use of the characteristics in CMP environments and multi-threaded workloads. Third, we propose a new router design for network-on-chip (NoC) based on a new memory technique. We introduce hybrid input buffers that use both SRAM and STT-MRAM for better performance as well as power efficiency.

Simulation results show that the proposed schemes improve the performance of off-chip networks through reducing the message size by 54% on average. Also, the schemes diminish the overheads of bounds checking operations, thus enhancing the overall performance by 11% on average. Adopting hybrid buffers in NoC routers contributes to increasing the network throughput up to 21%.

ACKNOWLEDGMENTS

First of all, I would like to thank my academic adviser, Dr. Eun Jung Kim, for her guidance. Her advises made it possible to complete my PhD study at Texas A&M University. I am truly grateful for her encouragement and constant motivation throughout this work. Also, I would like to thank my committee members, Dr. James Caverlee, Dr. Guofei Gu and Dr. Natarajan Gautam. Many thanks also go to Dr. Ki Hwan Yum for support throughout my PhD study, especially for his careful reading and comments on papers.

Second, I would like to thank all of the previous and current members of Dr. Kim's High Performance Computing Laboratory, especially Manhee Lee, Heung Ki Lee, Yuho Jin, Sungho Park, Lei Wang, Poornachandran Kumar, Rahul Boyapati, Jagadish Chandar Jayabalan, Wen Yuan, Hyunjun Jang, Rohan Kansal and Sagar Narayanan, for supporting and helping my research.

Last, but not least, I am especially grateful to my parents for their incredible support, patience and trust for me. Without their dedication and belief in me, I couldn't have completed this study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

1. INTRODUCTION

The demand for high performance computing (HPC) solutions has been constantly increasing since the computing environments were commercialized and popularly adopted by industry and government. In particular, with the current trend regarding cloud computing or big data processing as one of the hottest topics in an IT domain, designing efficient enterprise-level HPC systems in terms of both performance and power has been a highlighted issue that everyone pays attention to nowadays. So far, this demand was generally met with parallel processing systems, where a lot of processor chips are interconnected by off-chip networks such as Infiniband [1]. However, these previous mechanisms have a limitation on improving efficiency. In order to achieve a high degree of parallelism,a large number of processor chips must be connected and used, which increases the cost to build up systems significantly. Furthermore, integrating lots of off-chip components results in a huge amount of power consumption. Under a strict power budget, this cannot help being a critical limitation on system design.

Chip multiprocessor (CMP) has been successfully deployed on a commercial scale and widely adopted. CMP integrates multiple processing cores and cache banks on a single chip and shows better efficiency compared to the existing uniprocessor in terms of both performance and power. To this day, the number of cores and cache banks was not large and a shared bus was generally regarded as a communication

This dissertation follows the style of *IEEE Transactions on Parallel and Distributed Systems*.

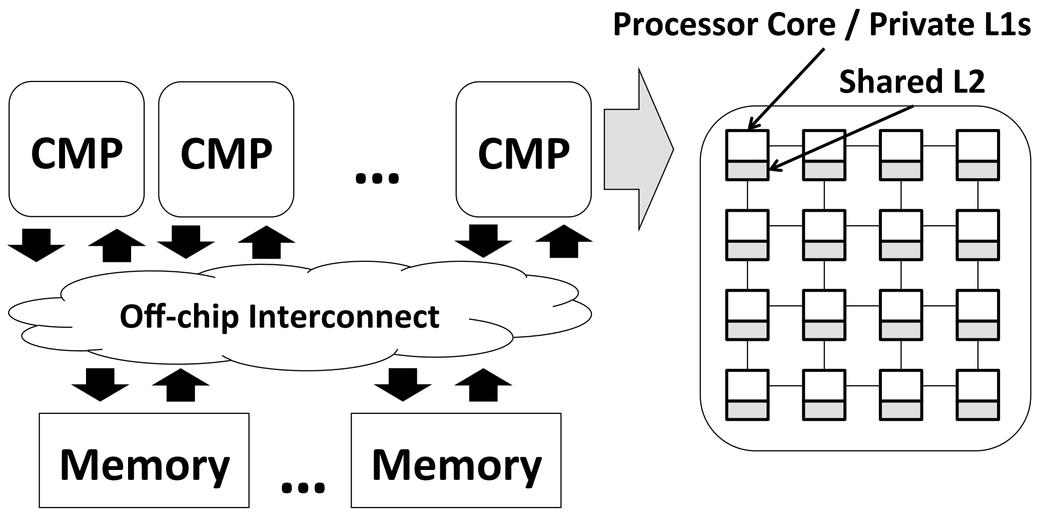


Fig. 1.1.: A CMP-based HPC System

medium providing enough bandwidth between nodes. But, as the technology trend moves to manycore design where a large number of cores, possibily more than a hundred, are fabricated in a chip, there is a growing need to introduce faster and more efficient interconnects.Network-on-chip (NoC) is being adopted as a new emerging solution of scalable interconnects. Accordingly, designing HPC systems using NoC-based manycore CMPs is beneficial compared to uniprocessor-based systems. In the case of large-scale systems for high performance requirements that a single CMP cannot satisfy, a multi-chip CMP systems can be built through connecting a number of CMP chips via off-chip interconnects.

Figure 1.1 illustrates a generic HPC systems with multiple CMP chips. A number of CMPs are interconnected with off-chip memories and each CMP contains multiple nodes each of which consists of a processor core, private L1 caches an d a part of shared L2 cache. All nodes in a CMP are connected through a mesh-style NoC.

This dissertation addresses a number of design issues regarding high performance multiprocessor systems based on CMP to achieve both performance and power efficiency. While performance and power efficiency is the first and foremost requirement in HPC, providing security cannot be neglected. Security issues are even more crucial in large-scale systems because the systems may be more vulnerable to attacks as they become more complicated due to abundant resources available to attackers. Therefore, security issues in multiprocessor systems are also considered in this dissertation.

This dissertation proceeds from macro to micro, starting from off-chip components to CMP and NoC design. First, we address the design issue of off-chip components including memory, last-level cache (LLC) and interconnects. Considering that networking overheads in multiprocessor systems are becoming one of the most influential factors in overall system performance, we attempt to reduce off-chip communication overheads through a data packet compression technique integrating a cache coherence protocol. We propose Variable Size Compression (VSC) scheme that compresses or completely eliminates data packets while harmonizing with existing cache coherence protocols. Also, we propose a hybrid counter management scheme that reduces the overheads of counter mode encryption mechanism.

Second, we propose fast and secure CMP design that protects systems from memory attacks with marginal performance overheads. Even though spatial safety of memory accesses is one of the main concerns for programs written in C/C++ to prevent runtime attacks and errors, most existing approaches either fail to solve the runtime overhead issues or cannot provide the complete solution for memory protection. Moreover, none of the previous work considers multi-threaded workloads running in multiprocessor systems. So we provide an architectural support for fast and efficient bounds checking for multi-threaded workloads in CMP environments. Bounds information sharing and smart tagging help to perform bounds checking more effectively by utilizing the characteristics of a pointer. Also the BCache architecture, a new cache and interconnect for efficient bounds checking, allows fast accesses to the bounds information.

Finally, a new design of NoC router based on a next-generation memory cell technology is proposed. Using high-density memories in input buffers helps to reduce the bottleneck through increasing throughput. Spin-Torque Transfer Magnetic RAM (STT-MRAM), a new non-volatile memory technology, can be a suitable solution due to its nature of high density and near-zero leakage power. But its long latency and high power consumption in write operations still need to be addressed. We explore the design issues in using STT-MRAM for NoC input buffers. Motivated by short intra-router latency, we use the previously proposed write latency reduction technique sacrificing retention time. Then we propose a hybrid design of input buffers using both SRAM and STT-MRAM to hide the long write latency efficiently. Considering that simple data migration in the hybrid buffer consumes more dynamic power compared to SRAM, we also provide a lazy migration scheme that reduces the dynamic power consumption of the hybrid buffer.

2. LOW-OVERHEAD AND SECURE OFF-CHIP INTERCONNECTS

2.1 Introduction

With the current trend in information technology, multiprocessor systems have been widely adopted to achieve better performance in diverse computing environments. Relying on traditional methods such as instruction-level parallelism (ILP) in uniprocessor environments has a limitation on performance improvement. Most software in recent days adopts multithread program structures, which were mainly used for high-performance computing environments exclusively. Under a high degree of parallelism, networking performance is getting more dominant than computation power in determining overall system performance. Since a traditional approach using shared buses is not a scalable solution, switch-based interconnection networks are regarded as a promising alternative in large-scale multiprocessor systems. Providing an effective solution to reduce communication overheads has been one of the major goals in the multiprocessor system design.

There has been plenty of research carried out to diminish overheads of interconnection networks in multiprocessor environments. This goal can be achieved in two ways: adopting a faster network with shorter latency and wider bandwidth, or decreasing the amount of network workloads. Some studies focus on the latter through data compression to reduce data packet overheads. Data compression used in memories and interconnects of multiprocessor systems helps to decrease the size of cache block data that is tranferred in the form of data packets through the network. Recently, a static data compression scheme [2] has been proposed to compresses data based on predefined fixed patterns, which fails to exploit dynamic communication behavior. To overcome this drawback, an adaptive compression scheme [3] uses tables containing frequently used patterns and updates them dynamically. However, both schemes simply reduce the packet size through compression and never attempt to incorporate the compression with underlying cache coherence protocols, which limits the amount of performance improvement at a certain level.

This research tries to take a step forward to overcoming the limitation. Here we attempt to reduce communication overheads through data packet compression that is aware of an intrinsic cache coherence mechanism. In the best case, the compression scheme enables data packets to be completely eliminated, which contributes to simplifying the miss handling procedure in coherent caches and reduces cache miss latency significantly. Data packets can be eliminated if the data pattern is known to the requestor without having to actually send and receive packets. It must be guaranteed that eliminated packets do not make any conflicts or errors with the existing cache coherence protocols. Also, the compression scheme used in multiprocessor systems must show good performance in handling a cache block whose size is small, not larger than 64 bytes in general.

We propose Variable Size Compression (VSC) scheme that is scalable and efficiently eliminates or compresses data packets. Packet elimination in VSC is done by managing matching status bits in the directory that denote whether each cache block data matches the most frequent pattern in the system. A reply message from the directory to the requestor carries the bit to notify that the requestor does not need to wait for a data message from the sender if the bit is set. If VSC fails to eliminate data packets, it divides the cache block into multiple units for compression. Unlike the existing schemes, VSC provides data compression with various unit sizes in parallel for more efficient compression.

Also, protecting systems from various malicious attacks has become another essential requirement in high-performance computing. Attackers may place a hardware device inside the system to steal information or purposefully modify system data. Moreover, institutes that handle confidential data and personal information necessitate a high level of security since any kind of information leakage could be extremely dangerous in mission-critical environments. Thus, providing enhanced security to protect computer systems from physical attacks is becoming one of the most important issues in modern computing. As the server markets move on to multiprocessor systems, new security issues unique to the multiprocessor environment are raised: protecting messages used for cache coherency. However, all the previous studies in multiprocessor systems [4], [5], [6], [7], [8], [9], [10] suffer from different amounts of performance overheads stemming from common underlying problems, longer/more messages. The longer and more messages caused by security mechanisms have been considered to be inevitable by all previous research because each message needs to carry extra data for security and additional messages are necessary to help or maintain the protection schemes. Although the performance implications of the longer/more messages were not investigated in the previous studies, it is quite straightforward to expect the longer/more messages to affect performance adversely.

For fast encryption/decryption, hardware security schemes [4], [6], [7], [8] have used a counter-mode scheme where a counter, not a plaintext, becomes an input of the security function to generate a one-time pad that will be later XORed with the plaintext to make a ciphertext. Therefore, if communicating parties can predict next counter values accurately and pregenerate pads ahead of time, they can encrypt or decrypt data by XORing with the pads without any delay. Due to the unpredictability of communication sequences in multiprocessor systems, there can be some mispredictions on counter values and they result in performance overhead, which is unacceptable in high-performance systems.

We propose a new hybrid counter management scheme that does not require embedding a counter value in a data packet. All previous schemes assume that each packet carries a counter value, which increases the message length by the counter size. We eliminate counters from data messages by enabling perfect prediction through a global counter scheme. We also use per-block counters to protect data stored in memory with a small storage overhead. Accurate prediction and low storage overhead can be achieved together with this hybrid scheme.

Simulation results show that the proposed scheme can reduce the message size by 54% on average and the overall performance by 23%, compared with the most recent compression scheme for interconnects proposed in [3].

The remainder of this study is organized as follows. We discuss related work in Section 4.2. In Sections 2.4 and 2.5, we explain the data compression technique and the hybrid counter management scheme in detail. Section 4.5 presents simulation results and analysis, and finally Section 5 summarizes our work and makes conclusions.

2.2 Related Work

2.2.1 Data Compression Mechanism

Various data compression techniques have been introduced to improve performance by increasing the capacity or reducing the latency [11], [12], [13], [14], [15], [2], [16], [3], [17]. Among them, a large number of hardware-based compression schemes implement dictionary-based compression algorithms [16], [14]. Dictionary-based schemes depend on building a dictionary which contains data words appearing in a message, and use the entries to encode repeated words. Significance-based schemes compress data by removing redundant information such as the high-order portion of address values or sign extension bits [11], [13]. Frequency-based mechanisms are based on the observation that small sets of values are found in load/store operations more frequently than other values [15], [17].

Data compression mechanisms have been adopted in diverse system domains. Cache or memory compression makes better use of the limited memory by compressing and storing data in a smaller space so that two or more compressed blocks can fit in a single block space [11], [16], [14]. Buses also can be utilized more efficiently through compression. Bus-Expander is proposed in [12], where frequent values of the most significant bits of the bus are stored in a table and used to efficiently cut the size of data in half. Even in on-chip interconnects, compressed messages can improve network utilization and system performance by reducing packet latency [2], [3].

2.2.2 Secure Processor Architecture

Several encryption and authentication schemes for secure uniprocessor systems were proposed in [18], [19], [20], [21], [22], [4], [9], [23], [7]. Caching or predicting counters have been explored to pregenerate pads before encrypted data blocks arrive at the processor from the memory [21], [9], [20]. Galois/Counter Mode (GCM) was adopted by Yan, et al. to accomplish authentication with encryption [4]. A per-block authentication scheme was first used in eXecute Only Memory (XOM) [18], but it is vulnerable to replay attacks. To overcome this vulnerability, a hash tree was proposed to authenticate memory and its performance overhead for authenticating all levels of the hash tree per memory transaction was reduced by authenticating a series of memory accesses at a later time. [23], [4], [7] With the popularity of multiprocessor architectures, new security issues that cannot be properly solved by uniprocessor security schemes became main research interests. Shi, et al. proposed a speculative execution mechanism [24] and Zhang, et al. exploited the characteristics of bus-based multiprocessor systems [10]. Rogers, et al. found that there is a strong locality of communication such that one processor communicates with a relatively small number of processors at a time [6]. Lee, et al. proposed I^2SEMS in which a global counter controller helps processors predict next counters more accurately [5]. Rogers, et al. proposed a single-level memory protection scheme to reduce the additional security translation overhead incurred by memory controllers [8].

2.3 Threat Model and Security Operations

First, we clarify which components in the system are assumed to be secure and which are not. The threat model assumed in this study is similar to those of other work which deals with security issues in multiprocessor systems. Every component in a chip, including processor core, registers, ALU and on-chip cache is assumed to be secure. We assume off-chip components, such as memory, data bus or interconnection network, to be insecure. Attackers can deploy special devices to the data bus or the interconnection network to eavesdrop and steal information during data transmission. Therefore, to guarantee confidentiality, data must be encrypted using cipher functions.

Before data messages are injected to untrusted components, they are encrypted using a block cipher function [1] with secret keys which are known only to the communicating pair. Encrypted data messages are decrypted at the receiver's side using the same cipher function with secret keys to restore original information. One problem with using encryption schemes is that encryption cannot begin until the entirety of the data becomes available. Consequently, encryption latency is added to the critical path. To hide the encryption latency, a counter-mode encryption scheme can be adopted. It uses counters to feed encryption engines, and precomputes one-time pads before the data becomes ready. Then one-time pads are XORed with actual data to obtain final ciphertexts. Since pads are generated in advance and XOR operation latencies are negligible compared with the encryption latency, we can reduce the encryption time. However, counter values must be predicted as accurately as possible to pregenerate one-time pads.

2.4 Variable Size Compression (VSC)

In this section, we explain how to eliminate packets and to compress and decompress data values in our framework. We propose a new compression scheme, which is called Variable Size Compression (VSC). Figure 2.1 summarizes the compression procedure of our scheme. With VSC, a cache block can be eliminated or divided into

[1]For encrypting data, block cipher functions are normally used. Stream cipher functions are known to be less powerful than block cipher mechanisms. 3DES and AES are well-known block cipher functions.

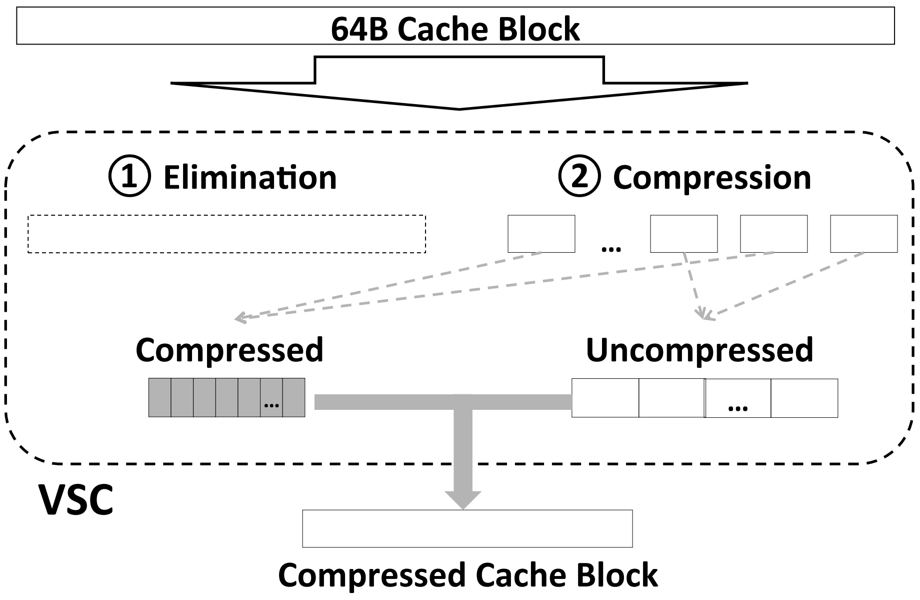


Fig. 2.1.: Compression procedure of a cache block

multiple small units to be compressed. After the compression, there can be compressed and uncompressed units, which will be explained in the following section. We assume that only data messages are compressed.

2.4.1 Efficient Data Compression Utilizing Variable Size Pattern Frequency

VSC uses two types of tables, compression and candidate tables, to compress or eliminate a cache block. Compression tables used in VSC contain most frequent data patterns in the network data traffic. After a cache block is divided into multiple compression units, each compression unit is compared to each entry in the compression table. If it matches one of the entries in the compression table, it is encoded into its corresponding index. If no entry is a hit, the unit is transferred uncompressed.

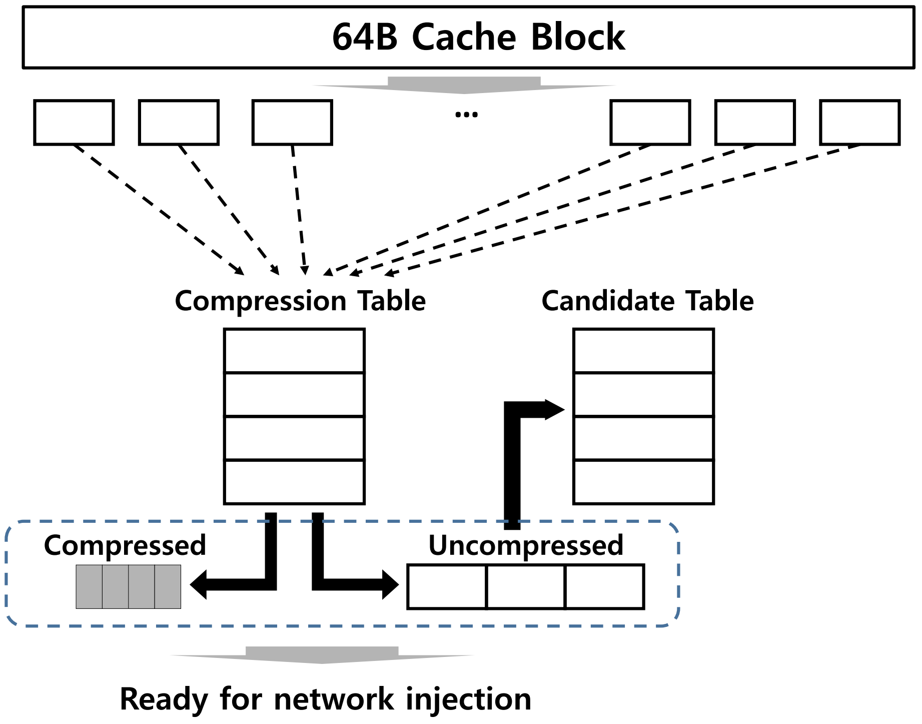


Fig. 2.2.: Compression and candidate tables

Each entry needs to count the number of hits to be used in the table update. In this manner, compression units become ready to be encrypted either in a compressed or uncompressed form. After the packet arrives at the receiver and is decrypted, compressed units are converted into original units by looking up the compression table at the receiver's side. Uncompressed units are directly delivered as they are. All nodes have the same contents in the compression tables so updating the compression tables is performed regularly in a centralized manner reflecting dynamic behavior of data patterns.

Candidate tables contain the most frequent values among uncompressed units and they are managed and updated dynamically and independently per node. Figure 2.2

describes how compression and candidate tables are used to compress cache block data. After compression, each uncompressed unit is compared to candidate table entries. If it matches one of the entries, the corresponding hit count of the entry increases. If not, one entry is evicted and replaced with the uncompressed unit data based on the Least Frequently Used (LFU) policy. Since accessing candidate tables is done after compression, the access latency is not in the critical path.

Updating compression tables is done as follows. First, hit counts of compression table entries and values/hit counts of candidate table entries in all nodes are gathered. Then hit counts of compression table entries for all nodes are summed up and all candidate table entries are merged and sorted in the descending order of total hit counts. Hit counts of the most highly ranked entries in the merged table are compared to those of compression table entries. If the hit count of a merged table entry is bigger than any of compression table entries' hit counts, the compression table entry is evicted and updated with the merged table entry. Finally, all hit counts of a newly updated compression table are initialized to zero to prevent aging effect and then the new compression table is broadcast to all nodes. Note that the update procedure is off the critical path, meaning that it can be done without affecting cache miss handling.

But in reality, the broadcast messages cannot reach all nodes at the same time due to the variation of packet transmission time. So there may be a synchronization issue between the two adjacent periods using different pattern sets, meaning that some nodes already receive the new pattern set while others do not. Here we handle this issue by using one-bit information, which is called a period bit (PB). A PB differentiates two adjacent periods by flipping the bit every time a new pattern set is ready. For example, if current period is represented by zero, the next period using a new pattern set can be represented by one. Each node maintains its own PB, updating the bit when it gets a new pattern set. Since the current period and the period after next are represented by the same bit value, some might think we should consider how to make difference between the two. However, it is a reasonable assumption that each period is long enough so that we do not need to be concerned about the issue. Also the amount of data traffic during the time when two pattern sets are used mixedly is quite minimal, which will be explained in detail in Section 4.5. Every time a cache coherence packet is injected into the network, the packet carries a sender's PB to let a receiver know the sender's status. Suppose that a sender of a cache block already got a new pattern set while a receiver did not. Then the sender must not use the new pattern for compression because the receiver cannot restore compressed data to its original one.

To minimize the packet size, we investigate how VSC can be applied, mostly the number of entries, each entry size, and the new cache-to-cache coherence packet structure. We ran 49 combinations of simulations with seven different entry sizes and seven different numbers of compression table entries in 8 and 16-processor systems. The results show that a four byte-long entry provides a very good compression ratio

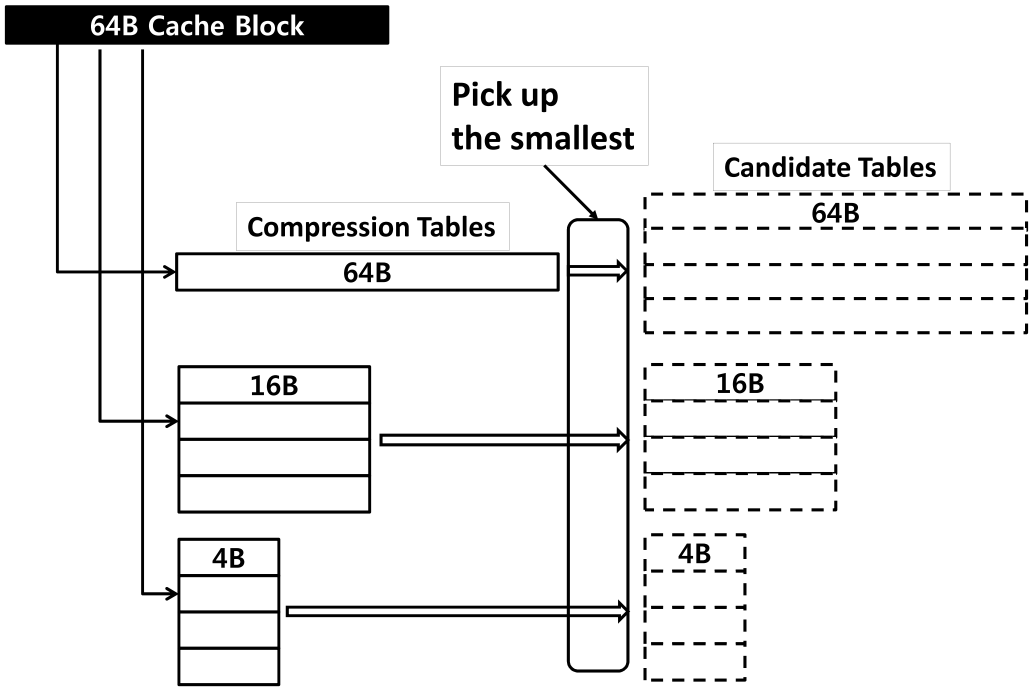


Fig. 2.3.: Data compression with various unit sizes

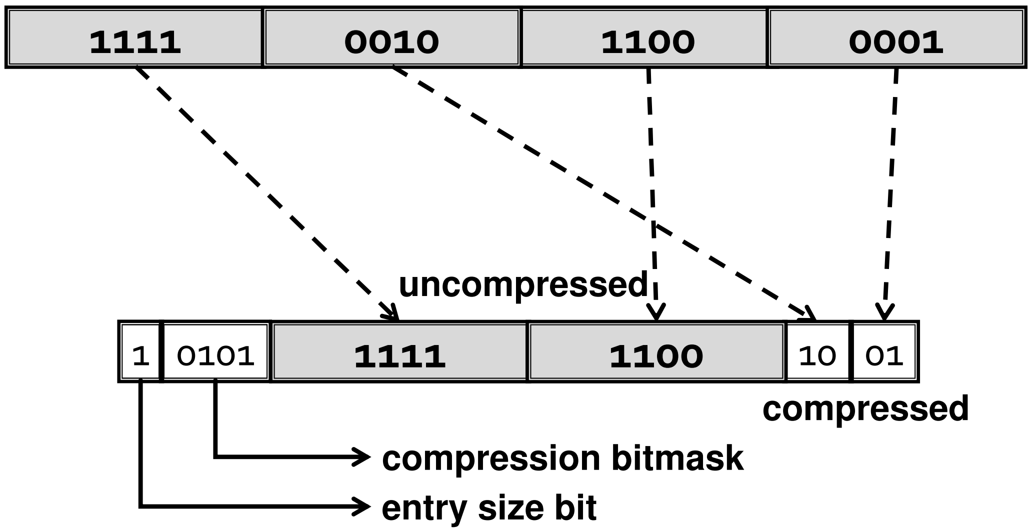


Fig. 2.4.: Compressed data block structure

only with four entries in a 64-byte cache block architecture. Since the entry size is four bytes and the index is just two bits long, assuming four entries per table, the unit size is reduced by a factor of 16. However, it would be beneficial to achieve a better compression rate if we try different unit sizes in every data compression and choose the best one. It is reasonable because VSC is free from the scalability issue, and adopting extra compression and candidate tables with different entry sizes is not problematic. So we use three different compression entry sizes: 4-byte, 16-byte and 64-byte as shown in Figure 2.3. For 4-byte and 16-byte, both compression and candidate tables have four entries. But, in the case of 64-byte that is the same as a cache block size, a compression table has only one entry, even though a candidate table has four entries as in the other two cases. The reason is that we do not need to use even index values once the cache block is compressed with 64-byte unit size, which means data could be completely eliminated. We will make a more detailed description of how it works in the next section. Also, each node maintains two sets of compression tables to handle two different pattern sets used in the adjacent periods.

To integrate VSC into a 64-byte block system, we divide a 64-byte block into a number of compression units, and compare each unit against the compression table. Then, each data packet should have additional information about which units are compressed. Figure 2.4 describes the structure of a compressed data packet, which depicts four 16-byte units belonging to one cache block. The content in each unit is displayed as a 4-bit number for simplicity, even though it should be actually 16 bytes long. An entry size bit that indicate the unit size used for compression is located at the head. If the bit is set, it means a 16-byte unit size is used. If it is clear, a 4-byte unit size is used. We do not have to adopt an extra bit for 64-byte unit size since a data packet itself could be completely eliminated and does not need to be sent. Note that all units in a cache block are compressed with the same unit size denoted by the entry size bit that shows the best compression rate. Then it is followed by a compression bitmask, which indicates whether each unit is compressed. The compression bitmask will be used by the receiver to restore the original data packet. Uncompressed and compressed units are located in order at the tail. In Figure 2.4, the receiver restores the original data block by getting the first and third uncompressed units from the packet and reading the second and fourth units from the compression table.

Since we divide one data block into multiple units, it takes much time to process each compression unit sequentially. Therefore, we adopt a number of duplicated compression tables to compress/decompress multiple units in parallel for each compression unit size. Latencies are overlapped through parallel compressions using duplicated tables. Note that we do not need to duplicate candidate tables because accessing candidate tables could be done off the critical path.

2.4.2 Selective Update in VSC

As explained in the previous section, updating the compression tables is performed in a centralized way. However, it may cause the scalability problem as the system size grows because all the information of compression and candidate tables must be collected from all nodes regularly to update the compression tables. It is already observed and well-known that the communication behavior in the interconnects of multiprocessor systems exhibits the high level of temporal locality. Also it is verified that only a few pattern dominates the total data traffic in overall [11], [2]. Therefore, we gather the information not from all nodes, but from some selected nodes that aggressively communicate with other nodes. The communication pattern between nodes looks asymmetric in many cases, meaning that some nodes send more packets than received packets and other nodes receive more packets than sent packets. Since each data packet is shown to both a sender and a receiver, it is a good way of reducing overheads to select only one of the two groups, aggressive senders or aggressive receivers. Here we get the information from aggressive senders transmitting data packets more than receiving them.

Selective update in VSC can be performed as follows. Each node i can be chosen to provide the information for update if

$$s_i > k \cdot r_i,$$

where s_i and r_i are the numbers of sent and received packets in node i, respectively. k is a constant for adjusting how selective VSC should be in choosing the nodes. As k becomes bigger, VSC becomes more selective and the number of chosen nodes decreases. It can reduce the update overheads, but it may worsen the compression ratio since the amount of gathered information also decreases. We will show a detailed analysis of the selective update in Section 3.5.2.

2.4.3 Data Packet Elimination Using Temporal Locality

While compression facilitates to reduce the data packet size, we can further eliminate data packets exploiting the frequent data patterns. Data packets can be completely eliminated using the compression table with only one 64-byte entry. Here we explain in detail how it is done. In previous section, we already described how the 64-byte compression table entry is shared by all nodes and how it is updated. We use extra one-bit information per cache block to denote whether each up-to-date cache block data matches the current table entry or not and it is assumed to be stored in a directory. We call it matching status bit (MSB) hereafter. In the case of fetching from disks, the memory becomes the owner. Since the directory is usually located along with memory, it is not a problem to keep track of most recent patterns for data blocks. However, if a store operation occurs, the processor becomes the owner. The processor compares the previous data pattern with the newly written one. If the two are different and one of them matches the compression table entry, it means the corresponding MSB should be updated and the processor notifies the directory. Note that the notification is needed only when the matching result changes after a store operation, and it is normally only a small fraction of total store operations. Detailed experimental results regarding this will be provided in Section 4.5. Using this method, the directory is able to keep track of the matching status of most recent data values for all cache blocks.

When a cache miss occurs, the processor sends a request message to the directory. Then the directory looks up the MSB of the corresponding requested block. If the MSB in the directory is clear, meaning that the block does not match the compression table entry, the directory follows the normal cache miss procedures to provide data to the requestor by forwarding the request to another owner node or sending the data block by itself if the memory is an owner. However, if the MSB is set, the directory sends a reply message to the requestor to notify that the data packet has been eliminated. Once the requestor receives the reply message from the directory, it does not need to wait for another message but only has to copy the compression table data to the corresponding cache line. The cache miss handling finishes at this moment, reducing the miss latency significantly.

As in the case of compression tables, the directory contains two MSBs for each cache block to handle two different pattern sets. The two bits are used alternately for each period by switching to the other bit every time a new pattern set is ready. Also, a valid bit should be assigned to each MSB to invalidate all MSBs used for the previous period when a new pattern set appears in the system. Thus, space overheads of MSBs and their valid bits are 4 bits per cache block, which corresponds to 0.7% of cache block size assuming 64-byte block. All control packets used for

Algorithm 1 Procedures for a requestor

```
Request to the directory with $PB[Req]$

procedure HandleReplyMsg
    if $MSB_{PB[Req]}$ is set then
        Fetch the data block from the frequent
        pattern for $PB[Req]$
    else
        Wait for a data msg
    end if
end procedure

procedure HandleDataMsg
    Restore the data block through decompression
    Store the data block in a cache
end procedure
```

Algorithm 2 Procedures for a directory

```
procedure HandleReqMsg
    if $Valid_{PB[Req]}$ & $MSB_{PB[Req]}$ then
        Reply to the requestor with MSB set
    else
        Reply to the requestor with MSB clear
        Forward to the owner with $PB[Req]$
    end if
end procedure

procedure HandleNotifyMsg
    Store the new MSB in $MSB_{PB[Own]}$
    Set $Valid_{PB[Own]}$
end procedure

procedure HandleDiskFetch
    Compare with two old/new frequent patterns
    Update $MSB_{old}$ & $MSB_{new}$
    Set $Valid_{old}$ & $Valid_{new}$
end procedure

procedure NewPeriod
    $MSB_{new}/Valid_{new}$ become $MSB_{old}/Valid_{old}$
    and vice versa
    Clear all $Valid_{new}$ bits
end procedure
```

Algorithm 3 Procedures for an owner

```
procedure HandleForwardMsg
    Compress with the pattern for $PB[Req]$
    Send the compressed data to the requestor
end procedure

procedure HandleStore
    Compare with the frequent pattern for $PB[Own]$
    if $MSB_{PB[Own]}$ needs update then
        Notify the directory
    end if
end procedure
```

packet elimination such as request, reply or notification messages must carry the sender's PB to handle two different pattern sets that may be used mixedly.

Algorithms 1, 2 and 3 describe the procedures of handling cache misses with the data packet elimination scheme. The period bits of a requestor, a directory and an owner are denoted by *PB[Req]*, *PB[Dir]* and *PB[Own]*, respectively. And the value of each period bit can be either *old* or *new*. MSB and its valid bit of two adjacent periods are denoted by $MSB_{old}/Valid_{old}$ and $MSB_{new}/Valid_{new}$.

2.5 Hybrid Counter Management Scheme

In this section, we propose a new hybrid counter management scheme that uses global and per-block counters together to enable perfect counter prediction with affordable storage overhead. First, we explain the advantages and disadvantages of the two counter schemes and how the hybrid scheme integrates them efficiently. Then we make a detailed description of how it works in a cache-coherent multiprocessor system according to the state of a cache block. Using the hybrid scheme, counter values are removed from data messages, thus reducing network overheads.

2.5.1 Per-block Counter vs. Global Counter

There are two types of counters used for counter-mode encryptions: a per-block counter and a global counter. In a per-block counter scheme, a system maintains a separate counter for each data block. The counter is incremented each time its corresponding data block is encrypted. A block address along with its counter value, which ensures the global uniqueness, is the input to the encryption function. The advantage of a per-block counter is that the counter size need not be large, since the counter is managed per block and a long wrap-around time is guaranteed even with a small counter size. However, it is difficult to predict the next counter value since we must maintain the counter information for each data block. In a global counter scheme, one global counter is maintained for the entire system. No additional information is necessary as input to the encryption function, because any two counters used in the system cannot be identical. The counter should be large enough, at least 64 bits, to prevent early wrap-around, which may cause a storage overhead problem.

Our hybrid counter management scheme takes advantage of the two schemes above. While adopting per-block counters for data blocks stored in memory to reduce storage overhead, we use a global counter for processor-to-processor or processor-to-memory communications to make it easy to predict the value accurately. The entire counter range is divided by the number of communication pairs and assigned to each of them. Each time a sender transmits a data message to a receiver, the counter value associated with the communication pair increases by one. A sender maintains the last counter value it used to send data to each receiver. Also, a receiver keeps track of the last counter value used by each sender so that it can predict the next counter value accurately. Thus, data packets do not have to carry counter values because

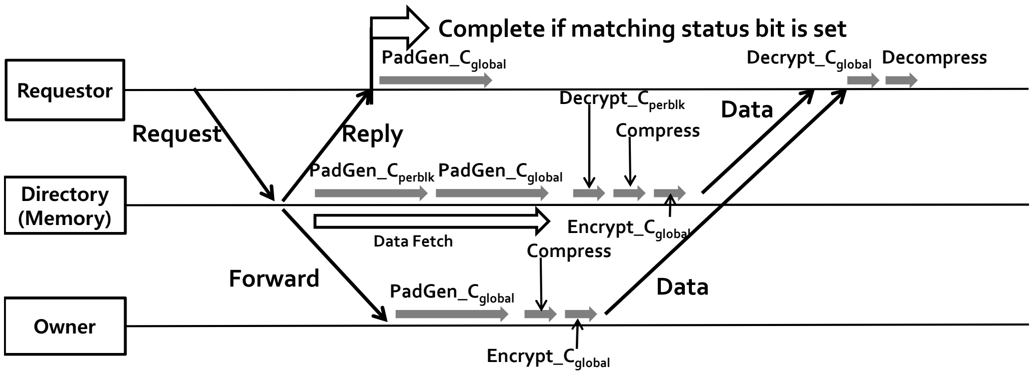


Fig. 2.5.: Handling a cache miss with the hybrid counter management

perfect counter prediction is guaranteed. A detailed description will be made in the next section.

2.5.2 Managing Counters with Perfect Prediction

Figure 2.5 depicts how a cache miss is handled with the proposed hybrid counter management scheme. We assume that the directory is located along with memory and its access time is shorter than memory access time. When a cache miss occurs in a processor, the processor generates a request message and sends it to the directory. Once the directory gets the request message, it immediately replies to the requestor with the cache block sender's id and the matching status bit explained in Section 2.4.3. Note that the information sent to the requestor can piggyback on the reply message used in the data packet elimination technique explained in Section 2.4.3. Thus, we do not inject an additional message into the network for this step. Cache misses are handled in different ways according to the cache block states.

Invalid or Shared State. If a cache block is in the invalid or shared state, the directory becomes the sender. In Figure 2.5, as soon as the directory receives a request message from a processor, it replies to the requestor by sending the directory's id along with the matching status bit of the corresponding block. The reply message does not contain a counter, since the requestor is able to predict a correct global counter value, C_{global}, only with the sender's id. Thus, we can decrease the reply message size by the counter length. As explained before, if the matching status bit is set, the miss handling finishes and the requestor only has to get the data from 64-byte compression table entry. Otherwise, the directory has to provide the requestor with data. The directory begins to generate two sets of pads from two counters: one set from a per-block counter (C_{perblk}) used to decrypt the block stored in memory, and the other from a global counter (C_{global}) assigned to the current communication pair that is used to encrypt the data before transmission. Definitely, the global counter value used here is the same as the one the requestor will predict. Concurrently the directory accesses memory to fetch the data block. Since the memory access time is long enough to cover the generation time of the above two sets of pads, the pad generation time is not in the critical path. When the data fetch is complete, the data block is decrypted with pregenerated pads from C_{perblk} [2]. Then it is compressed by VSC and encrypted with pads from C_{global} to be sent off to the requestor. Since pads are already available when the data fetch completes, decryption and encryption can

[2]We encrypt data blocks stored in memory using a per-block counter scheme as in [4] when they are written back to memory.

be done without delay. Note that a data message does not have to embed a counter because the requestor is able to predict the value accurately.

Once the requestor gets a reply message from the directory, it first checks up the matching status bit in the message. If the bit is clear, the requestor begins to generate pads used for decrypting the incoming data block. The requestor is informed of the sender, so it can predict the counter value, C_{global}, because it keeps track of the last counter value from the sender; the directory in this case. When the data message arrives, the requestor decrypts it using the pregenerated pads and decompresses it to restore its original information.

Modified or Owned State. If a cache block is in the modified or owned state, a different owner node provides the requestor with the data. In Figure 2.5, after the directory receives a request message, it replies to the requestor with the owner's id and the matching status bit. If the matching status bit is clear, the directory forwards the request message to the owner node. As soon as the requestor gets a reply message from the directory, it can generate pads for decryption from a global counter since it is notified of the owner's id. Once the owner receives the forwarded message from the directory, it generates pads from a global counter, C_{global}. Since the cache block is located in the owner, not in the memory, a per-block counter is not required. In order to reduce the pad generation delay from the owner, each processor maintains pregenerated pads with next counter values for its destinations. Since our global counter scheme is address-independent, pregenerated pads can be used for any data blocks in a specific node. In our simulation, just one counter for each destination turns out to be enough. Also the communication pattern shows high temporal locality as mentioned in Section 2.4.2. Therefore, each node caches a pregenerated pad for each of four recently communicated destination nodes. The owner compresses and encrypts the data block with pregenerated pads using C_{global} in a short time and sends it to the requestor without a counter value. When the data block arrives at the requestor, it is decrypted using the pregenerated pads and decompressed.

Providing Security for Non-Data Messages. Up to now, we have considered only data messages as the target of encryption/decryption and authentication using our scheme. For non-data messages including request, forward or invalidation messages, our scheme protects them in the same way. If we have to protect only data messages, pads do not need to be maintained because they can be generated on demand after a requestor is replied. However, to protect non-data messages, each node has to keep pregenerated pads for its all communication counterparts since it does not know from which node it will get a message. Thus, we can protect all kinds of messages without delay, sacrificing some space for pregenerated pads. Note that counter values are not transmitted through the network in either case. Data integrity also can be guaranteed using MACs.

2.5.3 Implementation Issues

Even though our assumption in the previous section is that the directory is located along with memory, the hybrid counter scheme can be used with a system where the directory and memory are separated as well. In this case, the directory forwards the request message to the memory as in the case of handling modified or owned cache blocks.

The hybrid counter scheme has two kinds of storage overheads: one for per-block and the other for global counters. Since per-block counters are small, they do not incur big overheads. To use a global counter for communications, processors and memories need to keep track of the last counter values used. Thus, each node must have room to store counter values for each communication direction for all destinations.

The hybrid counter management scheme assumes a system using a directory-based cache coherence protocol and a point-to-point interconnection network. It cannot be used along with snoop-based and token-based protocols, or multicast-based networks. As the system size keeps growing, more systems use switch-based networks with directory-based protocols. Bus interconnects using snoop-based protocol are not suitable for large-scale systems because of the scalability issue.

In-order packet delivery must be guaranteed to make the hybrid counter management scheme work. When messages arrive out of order, they can be detected in the data authentication stage. Out of order messages incur authentication failures because their MACs are generated using different counter values from what a receiver predicts. If the degree of out-of-order is not serious, it can be handled by placing a small buffer in each communication node to store a few incoming messages so that messages in the buffer can be reordered correctly.

2.6 Performance Evaluation

2.6.1 Simulation Framework

We measure the performance of the proposed schemes using Simics full-system simulator [25]. In order to simulate a shared memory model for a multiprocessor system, we also use General Execution-driven Multiprocessor Simulator (GEMS) [26] which is implemented in the form of a module used by Simics simulator. We use MOESI_SMP_directory as a cache coherence protocol, which has five cache block states and provides directory-based cache coherency. Table 3.3 shows the system parameters used in the simulation. Also we use AES as an encryption mechanism and its latency is configured as 80 cycles, which is comparable to 80ns in a 1GHz processor [27]. Since we assume that each processor has its own off-chip memory, the total number of nodes in the system is double the number of processors.

Benchmarks used in this study are three SPLASH-2 [28] (*radix, fft, lu*), four SPEC OMP2001 (*equake, fma3d, swim, mgrid*) and two PARSEC [29] (*streamcluster, swaptions*) benchmarks for scientific workloads. We also use SPECjbb2000 server benchmark for commercial workloads.

Table 2.1: System parameters

Parameters	Values
CPU	1GHz UltraSPARCIII+, 8/16 processors
L1 I & D cache	4-way, 16KB, 3 cycles
L2 cache	4-way, 4MB, 12 cycles
Cache block size	64B
Memory	4GB, 480 cycle access time
Number of memory modules	same as the number of processors
Directory access time	80 cycles
Network topology	hierarchical switches (fanout degree of 4)
Network link bandwidth	3G bytes/sec.
AES latency	80 cycles pipelined with 5 cycle occupancy
OS	Sun Solaris 9
Compression/ decompression latencies	2 cycles

2.6.2 Simulation Results

First, we clarify how much VSC is useful in data compression by measuring average compression ratios, which denotes the average of original block size divided by compressed block size. The performance of VSC is compared to the most recent two compression schemes [3], [2] for interconnects. Figure 2.6 shows the average compression ratios for three different compression schemes: table-based adaptive compression called Frequent Value Table (FVT) [3], significance-based static compression called Frequent Pattern Compression (FPC) [2] and VSC. VSC performs as good as the other two schemes in overall, while greatly surpassing them in *mgrid-8p* benchmark.

In Figure 2.7, we show the overall system performance in terms of instructions per cycle (IPC) of three different schemes: baseline, FVT and VSC. The baseline

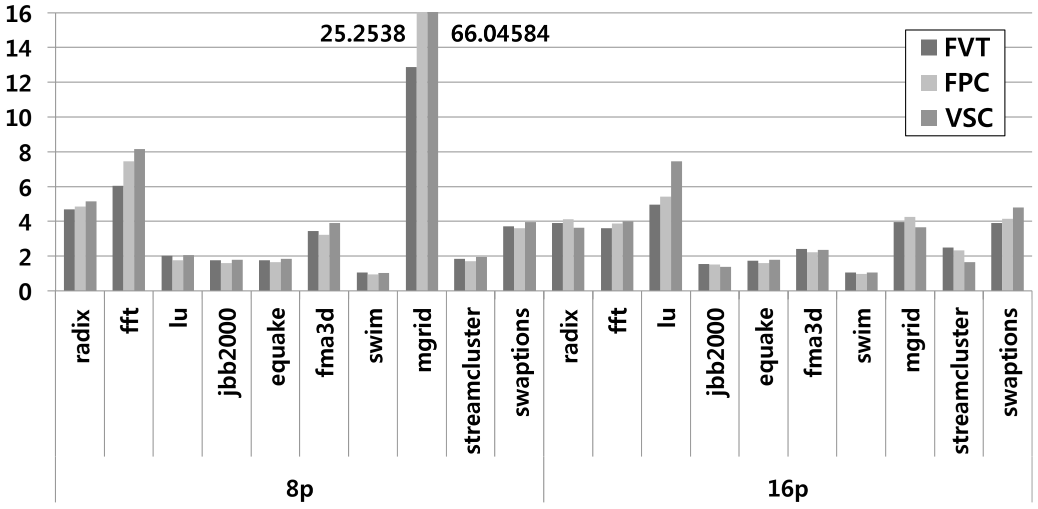


Fig. 2.6.: Compression ratios

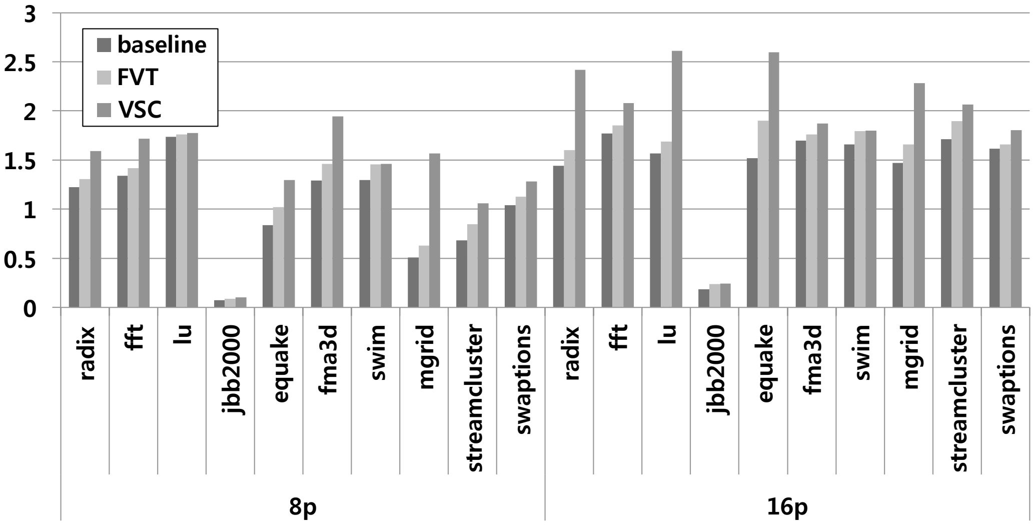


Fig. 2.7.: Overall system performance

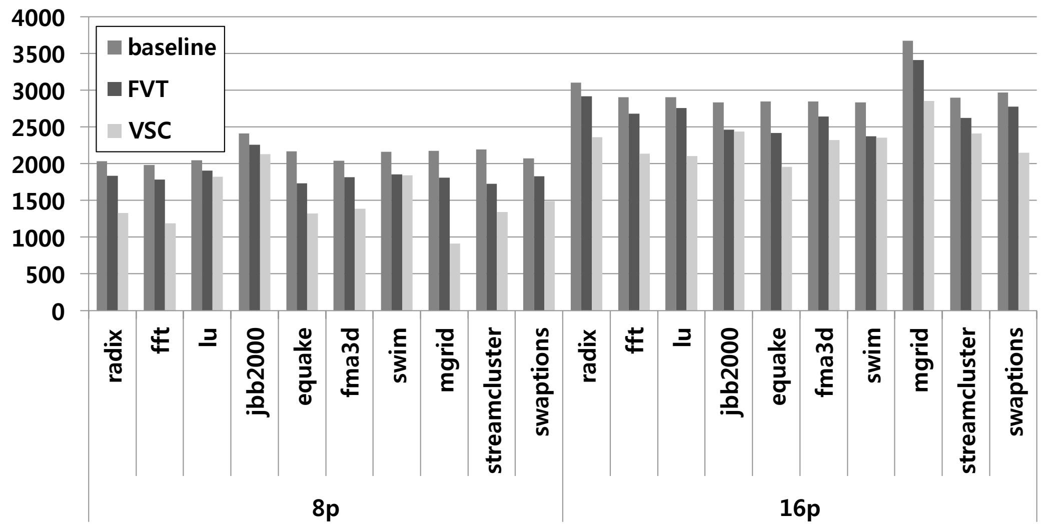


Fig. 2.8.: Average cache miss latency

system does not use compression techniques for data packets, while FVT and VSC try to compress data packets to reduce their sizes. VSC shows better performance than the others, by 36% compared to the baseline and 23% to FVT on average. This enhancement is achieved because of the significant reduction of cache miss latency by the compression and packet elimination scheme in VSC. To support our analysis, we measure average cache miss latency of the three schemes as shown in Figure 2.8. Again, VSC outperforms the other two schemes, by 34% compared to the baseline and 20% to FVT on average. By comparing the results of FVT and VSC, we can find out that the amount of enhancement looks quite larger than that in Figure 2.6. This is because the packet elimination in VSC greatly helps to simplify a miss handling procedure in addition to packet size reduction, by cancelling request forwarding and data packet transmission. FVT compresses a data packet through encoding the data into the index of a table entry it matches, which reduces data packet latency but cannot totally hide it. But in VSC, the most frequent 64-byte data pattern is known to all nodes in the system, and data packet transmission is completely cancelled if the requested data matches the pattern because the data is already available to the requestor.

In order to analyze the results in more detail, we examine how much VSC affects network latency compared to the other two schemes. Figure 2.9 shows average data packet latency, which is a network delay of a data packet. Here we assume that the latency of eliminated packets in VSC is zero. VSC achieves approximately 64% of enhancement from the FVT scheme, even decreasing the latency by more than half on average compared to the baseline. By eliminating and compressing data packets, VSC makes a significant contribution to reducing network overheads, enhancing the overall performance.

We measure the amount of eliminated data packets in VSC to clarify the effect of packet elimination. As shown in Figure 2.10, up to 96% of data packets can be eliminated depending on the workload and 41% of packets are removed on average. Table 2.2 illustrates numbers of MSB updates by owner nodes. As explained in Section 2.4.3, the processor must notify the directory if it needs to update MSB when it becomes the owner after a store operation. Since it could incur considerable network overheads if the notification occurs frequently, we measure the percentage of notifications over the total number of store operations. Results show that the average percentage is approximately 2.8%, which is marginal in overall.

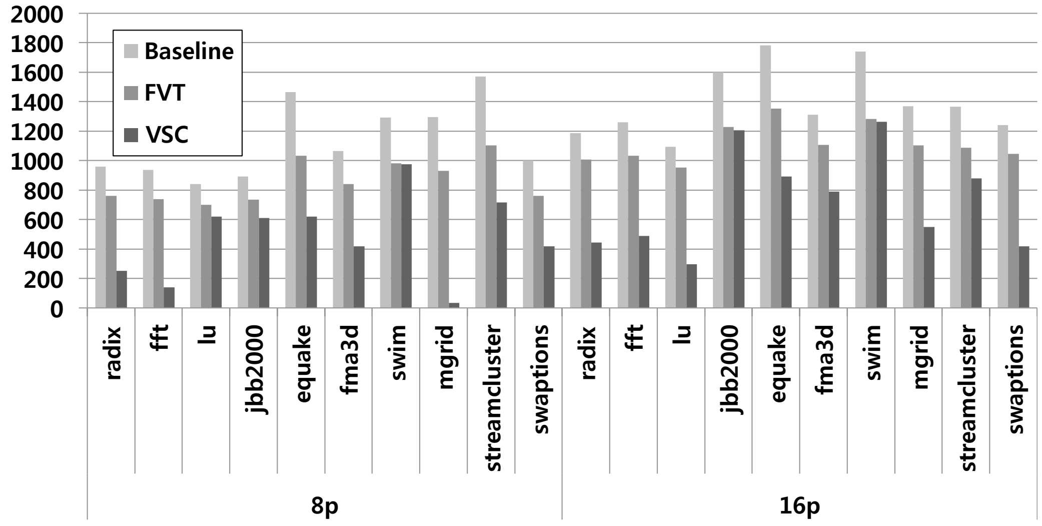


Fig. 2.9.: Average data packet latency

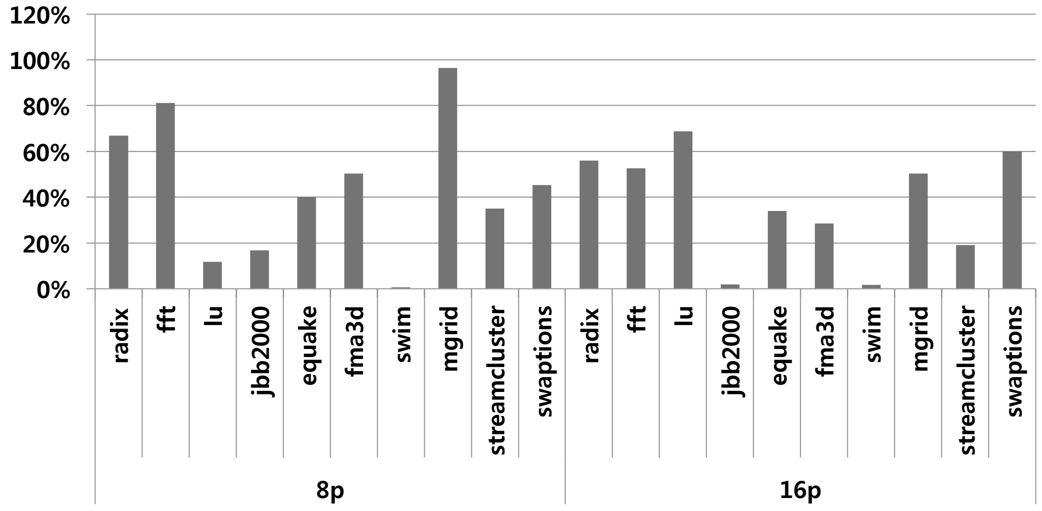


Fig. 2.10.: Eliminated data packets

Table 2.2: The percentage of notifications from owner nodes

Processors	Benchmark	# of Stores	Notifications	Percentage
8p	radix	128857	1536	1.192019%
	fft	102629	1657	1.614553%
	lu	33893	330	0.973652%
	jbb2000	408706	13229	3.236801%
	equake	848430	8015	0.944686%
	fma3d	1546842	149257	9.649143%
	swim	2401947	61208	2.548266%
	mgrid	13677	483	3.531476%
	streamcluster	9259	25	0.270008%
	swaptions	24932	332	1.331622%
16p	radix	98063	1193	1.216565%
	fft	102629	1657	1.614553%
	lu	312470	4752	1.520786%
	jbb2000	114726	1207	1.052072%
	equake	320219	5172	1.615145%
	fma3d	675611	98320	14.552753%
	swim	713277	1511	0.211839%
	mgrid	21951	1354	6.168284%
	streamcluster	6578	31	0.471268%
	swaptions	9588	127	1.324572%

Table 2.3 shows how many data packets are injected during the time when two pattern sets are mixedly used between two adjacent periods. As seen in the table, the amount of those packets among total number of injected packets is less than 0.05% on average, which is an almost negligible result.

We measure the compression rates with various update periods from 1 to 16 million cycles to figure out how much VSC is affected by the update period. Figure 2.11 shows the compression rates normalized to the result with 1 million cycle period. It confirms that the compression rate decreases by approximately 15% on average

Table 2.3: Mixed packets in two adjacent periods

Processors	Benchmark	Total	Mixed Pkts	Percentage
8p	radix	239156	115	0.048086%
	fft	658918	274	0.041583%
	lu	73898	53	0.07172%
	jbb2000	492910	129	0.026171%
	equake	4330889	1099	0.025376%
	fma3d	2635812	930	0.035283%
	swim	6430531	2307	0.035876%
	mgrid	16424012	4172	0.025402%
	streamcluster	2102753	1192	0.056688%
	swaptions	4001181	1273	0.031816%
16p	radix	234829	104	0.044288%
	fft	124391	66	0.053059%
	lu	568546	310	0.054525%
	jbb2000	487468	75	0.015386%
	equake	1054687	691	0.065517%
	fma3d	997459	577	0.057847%
	swim	1070339	615	0.057458%
	mgrid	152313	38	0.024949%
	streamcluster	503260	285	0.056631%
	swaptions	3210054	1236	0.038504%

as the period increases up to 16 times. Both 8-processor and 16-processor systems show the similar results, which proves that VSC is scalable with respect to the system size. We also measure the compression ratios and the number of selected nodes with different k values to clarify how the selective update is beneficial compared to the non-selective one. Figure 2.12a shows the compression ratios normalized to the result with a non-selective method. As shown in Figures 2.12a and 2.12b, we can verify that the number of selected nodes reduces significantly by 70% on average with the sacrifice of less than 1% in terms of a compression rate. It means that the

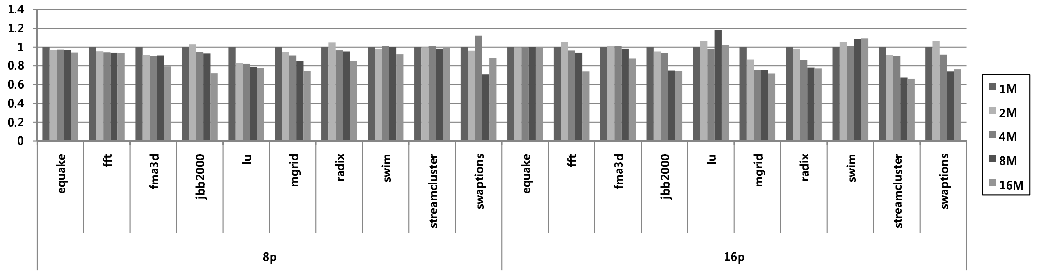
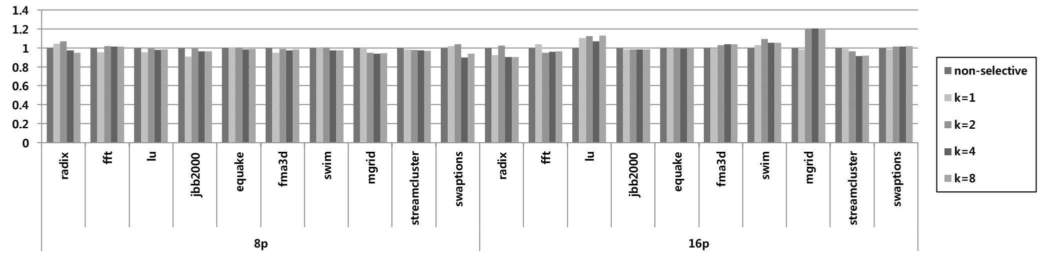
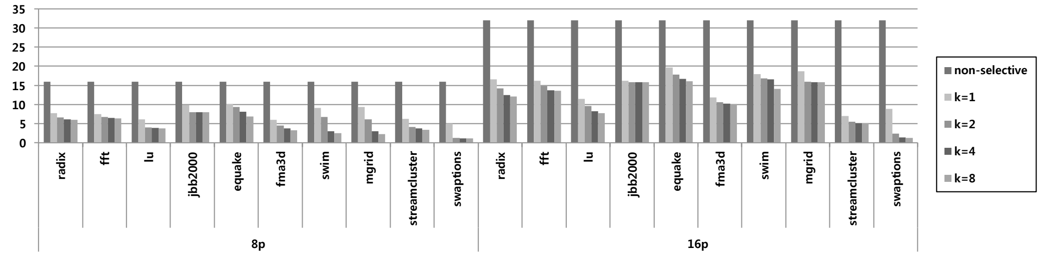


Fig. 2.11.: Compression rates under various update periods



(a) Compression rates

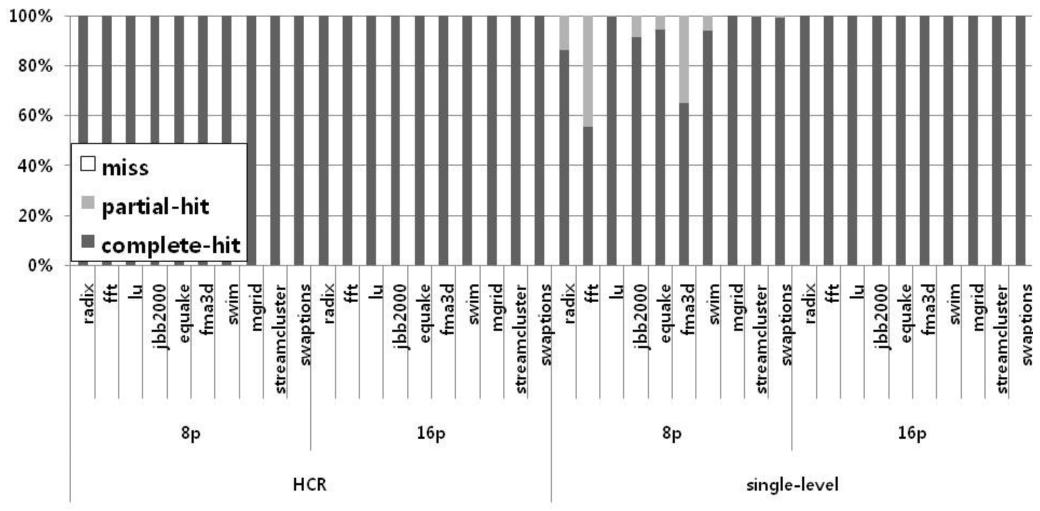


(b) Number of selected nodes

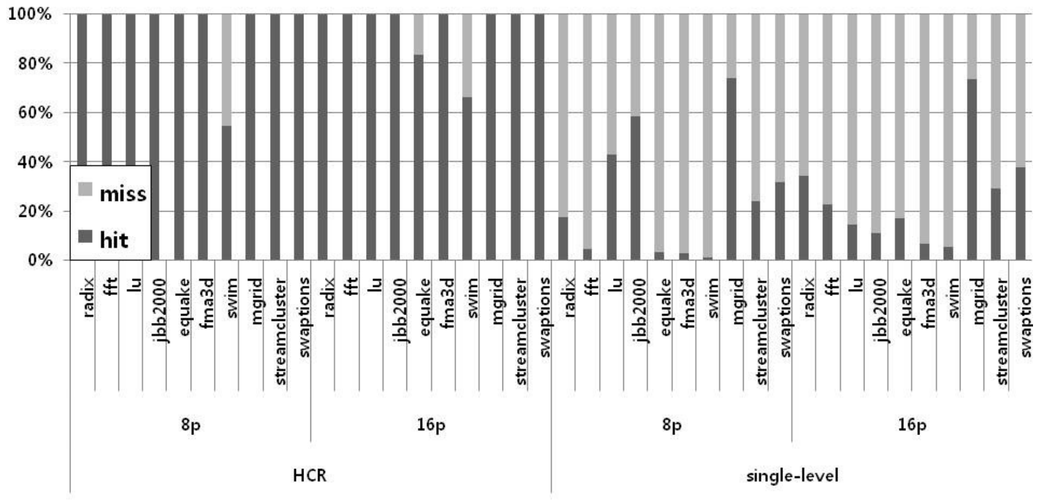
Fig. 2.12.: Effect of selective update

network overheads can be considerably reduced by 70% for gathering information from nodes in VSC update, with almost no penalty in compression rates.

To verify the effectiveness in terms of security overheads, we examine how efficient our scheme is in terms of hiding encryption and decryption latencies by looking at the pad hit and miss rates. Here we compare our scheme with the most recent work in secure multiprocessor architecture: the single-level scheme proposed in [8] where



(a) Requestor



(b) Sender

Fig. 2.13.: Pad hit and miss rates

each processor has a 32-entry owned-block pad buffer and a 32KB counter cache. In the single-level scheme, the directory replies to the requestor with a counter value as soon as it receives a request message. Our scheme is denoted by HCR, since it consists of Hybrid counter management, data Compression and packet Removal. Figure 2.13a shows the percentage of pad hits and misses in the requestor's side for each benchmark and compares the two schemes. A complete hit means that pads are fully generated before a data message arrives. Likewise, a partial hit means that pads are partially generated and a miss means pads are not generated. In both schemes, the requestor gets a reply message from the directory so that it can begin pad generation in advance. In our scheme, the directory is able to respond to the requestor immediately, because it only has to notify the sender's id. Whereas in the single-level scheme, the directory must carry a counter value in the reply message. If the counter value is not found in the counter cache, the directory has to access memory to fetch it, which postpones the transmission of the reply message. Figure 2.13b shows pad hit and miss rates in the sender's side. As in the receiver's case, a sender can reduce encryption latency if pregenerated pads are available. In our scheme, a sender maintains pregenerated pads from the next counter value for each destination. The pads may not be available when many requests arrive concurrently. If so, since pregenerated pads are used up quickly, encryption latencies are added to the critical path. In the single-level scheme, each processor maintains an owned block pad buffer which stores pregenerated pads for modified data blocks. If a sender fails to get pads from the buffer, it cannot hide the encryption latency. Again, our scheme shows almost perfect pad hits, whereas the single-level scheme does not. A pregenerated pad in our scheme can be used for any data packet to the same destination. However, in the single-level scheme, each pad is only for its corresponding data block.

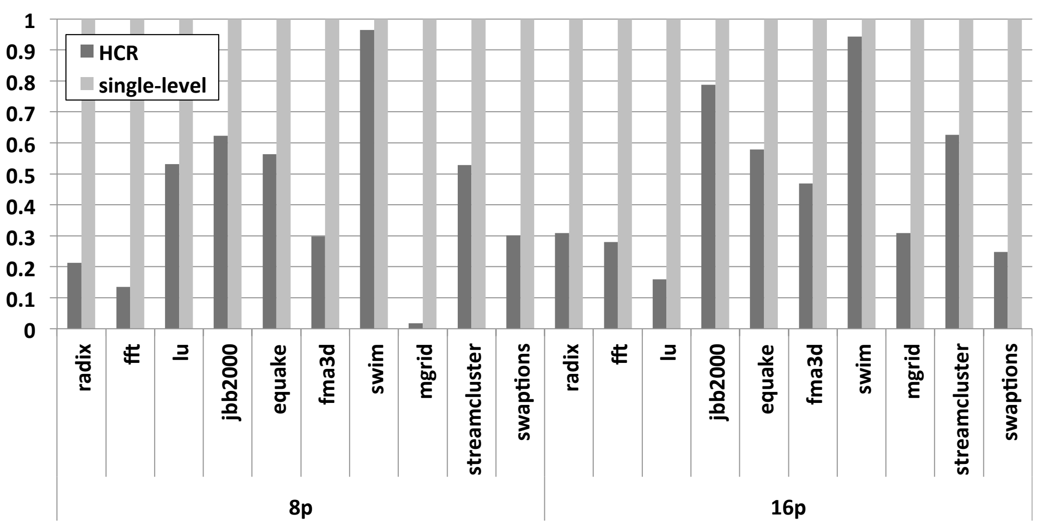


Fig. 2.14.: Encryption overheads

Finally, we investigate how much our scheme reduces encryption overheads compared to the single-level scheme. Figure 2.14 depicts encryption overheads, which mean total number of pads needed to encrypt data messages. As we have shown in this section, our scheme efficiently reduces the packet size or completely eliminates packets, which leads to the reduction of total number of pads needed. It reduces encryption overheads by 56% on average.

2.7 Conclusions

In this study, we have proposed VSC to alleviate network overheads by reducing the number of packets as well as the packet size. The packet compression technique, VSC, coupled with an underlying cache coherence mechanism, achieves significant performance improvement by cancelling packet transmission for the most frequent data known to all nodes. Eliminating data packet transmission tremendously reduces cache miss latency, which enhances overall system performance. The hybrid counter management achieves perfect counter prediction with low storage overhead by using global and per-block counters together, which allows data packets not to carry counters for encryption. In simulation results, VSC outperforms the existing FVT compression scheme [3] by 23% on average in terms of overall execution time, and by 20% in cache miss latency. Compared to the baseline system with no compression techniques, the amount of performance improvement in VSC significantly increases up to 36% in terms of overall execution time and 34% for cache miss latency.

Our work can be explored further by investigating the performance effect when using different topologies for multiprocessor systems. Also, we plan to analyze the performance of VSC under mixed workloads or different system domains such as chip multiprocessor (CMP) systems.

3. EFFICIENT MEMORY PROTECTION SCHEME FOR CMP SYSTEMS

3.1 Introduction

The C/C++ programming languages have been widely used in various programming environments since they were first introduced. However, the lack of support for spatial safety of memory accesses in C/C++ has been constantly addressed as one of the major drawbacks. Pointers and array indices must be properly managed to access the memory within their bounds originally assigned to them. If they are allowed to go out of bounds, they might be used as a means of software attacks by accessing unpermitted memory areas, which worsens the vulnerability of a system. The problem might become worse when we consider that C/C++ are dominant in the system programming area, especially for operating systems and middlewares performing mission-critical tasks. Not only the runtime memory attacks, but pointers and arrays that are not properly checked by programmers cause lots of memory errors in the software written in C/C++, which increases the debugging cost significantly.

Moreover, it is obvious that unsafe memory accesses can cause far more complicated and dangerous situations as multi-core/multi-threaded programming environments become widely adopted in various application domains. In a program running multiple threads, for instance, a pointer locally declared in a thread function may exist as multiple instances since each thread has its own stack. Or, a number of thread-specific pointers may point to one globally accessible memory object by copying a global pointer to multiple thread-specific pointers. It is common in multi-threaded programs to make a global object shared by threads and synchronize the accesses through mutexes or semaphores. The increased code complexity in multi-threaded software makes it even more difficult and expensive to find out memory access errors and to debug the program. Therefore, it is crucial for chip-multiprocessor (CMP) systems running multiple threads to be equipped with the mechanisms that can effectively protect all the memory accesses of multi-threaded workloads.

A number of schemes have been proposed in order to handle unsafe memory accesses. However, most of the existing schemes solely rely on software-based approaches, which inevitably incurs significant runtime overheads [30], [31]. Some hardware-based schemes that provide architectural supports have been suggested, but many of them are specialized for some specific types of memory attacks, not covering the whole [32], [33]. HardBound [34] is the first work to provide an architectural support for efficient bounds checking operations and gives a more general solution to protect a system from a wide range of memory attacks. However, it is designed for uniprocessor systems with single-threaded programs, which does not cover the issues of multi-threaded workloads running on CMP systems. Not only does it waste memory space by duplicating bounds information that can be shared by multiple pointers, but also it increases overheads by performing unnecessary bounds checking for pointers that are already verified to be safe. Moreover, HardBound does not consider any hardware support for CMP systems. Even though HardBound can be deployed in CMP systems as well, it incurs performance overheads due to the inefficient management of bounds information.

In this study, we propose an efficient bounds checking mechanism that provides an architectural support with marginal area and performance overheads in CMP systems. We take advantage of the unique characteristics of cache coherence mechanisms and multi-threaded workloads running on CMP architecture. First, to reduce the space overheads of bounds checking, we present bounds information sharing. When multiple pointers are referencing the same memory object, they can share the bounds information of the object as well through an architectural support for sharing bounds information using bounds data addresses.

Next, we introduce two schemes to reduce the performance overheads of bounds checking: *Smart Tagging* and *BCache*. Smart Tagging avoids unnecessary bounds checking that can increase extra overheads. It can be easily observed that a pointer is kept safe most of the time, implying that its bounds do not need to be checked. We make the best use of these characteristics of a pointer to eliminate unnecessary bounds checking via Smart Tagging.

BCache is a new cache and interconnect architecture that facilitates fast access of bounds information. BCache allows duplications of the same bounds information in multiple L2 caches, which reduces the access latency. Performance overheads may increase when bounds information located in multiple places is frequently updated or invalidated. But we observe that the bounds information does not change frequently even though its associated pointers keep changing, and the overheads can be effectively managed. We also explore the BCache architecture design in large-scale CMP systems under state-of-the-art and most frequently used design trend such as the concentrated mesh (CMesh) topology, nanophotonics and 3D stacking. Moreover, we show that BCache improves performance when it is used to store regular read-only data as well as bounds information to prove the versatility of BCache, making compensation for its hardware cost.

Simulation results show that the bounds information sharing reduces the space overheads by 15% on average. Also, we observe that 98% of bounds checking can be skipped through Smart Tagging and the average miss latency of bounds information decreases by 49% on average using BCache. Eventually, these improvements enhance the overall performance in terms of the number of clock cycles per micro-operation (CPμ) by 11% on average when all the memory operations are executed. Total energy consumed by BCache also decreases by 47% and 61% in caches and interconnects, respectively.

The remainder of this study is organized as follows. We discuss related work in Section 4.2. In Sections 3.3 and 3.4, we explain the bounds information sharing, Smart Tagging and the BCache architecture in detail. Section 4.5 presents simulation results and analysis, and finally Section 5 summarizes our work and conclusions.

3.2 Related Work

Several schemes have been proposed regarding the secure multiprocessor design to protect systems from various kinds of attacks. A large portion of them deals with architectural supports to prevent physical attacks during the data transmission through vulnerable parts of the system, mostly off-chip memories and interconnects [6], [5], [8]. Also, there have been studies to solve the security problems associated with new computing technologies [35], [36].

The protection of spatial memory errors has been explored for a long time in order to prevent memory access violations in programs written in programming languages that do not support boundary checking. A number of schemes based on software approaches were introduced to enforce spatial memory safety. In [37], [38], [39], fat pointers were used to associate each pointer with its bounds information needed for bounds checking. Nethercote [40] attempted to generate fat pointers using dynamic binary instrumentation. Also Heap Server [41] was introduced to protect the heap metadata. Nethercote and Seward [31] supported shadow values to be used for tracking and detecting dangerous memory accesses. Hastings and Joyce [30] used a red-zone for each allocated memory to detect bounds errors. Berger and Zorn [42] carried out probabilistic analysis to guarantee memory safety using approximated infinite heaps. In [43], different software-based implementations of bounds checking were analyzed, and some taint-based optimization techniques were introduced to reduce runtime overheads. Some recent studies provided more efficient bounds checking mechanisms with less overheads [44], [45], [46], [47], [48], [49]. In [47], [48], algorithms that eliminate redundant bounds checkings were proposed, and Chuang *et al.* [49] reduced the amount of memory space needed by metadata for bounds checking. Although these studies contributed to preventing spatial memory errors, most of them suffered from huge runtime performance overheads since they were software-based approaches. Moreover, they provided insufficient analysis of overall system performance with multi-threaded workloads.

To overcome the overhead issues of software-based schemes, making an architectural support for memory safety began to be considered. Venkataramani *et al.* [33] used hardware support to check up the validity of memory block data. Shao *et al.* [32] provided an architectural support of bounds checking for arrays and pointers. However, none of the approaches mentioned above dealt with multiprocessor systems and multi-threaded workloads. HardBound [34] is a hardware-based approach for bounds checking while maintaining the software overheads minimal. It accesses pointers and their associated bounds information with hardware support and propagates them through the system. Also it automatically performs bounds checking before pointer dereferences. But, HardBound always maintains a separate copy of bounds information per pointer, even when the bounds information can be shared by multiple pointers. Furthermore, it performs bounds checking for all pointer dereferences, no matter whether a pointer value is already proved to be safe or not. The biggest limitation of HardBound in CMP systems is that HardBound does not

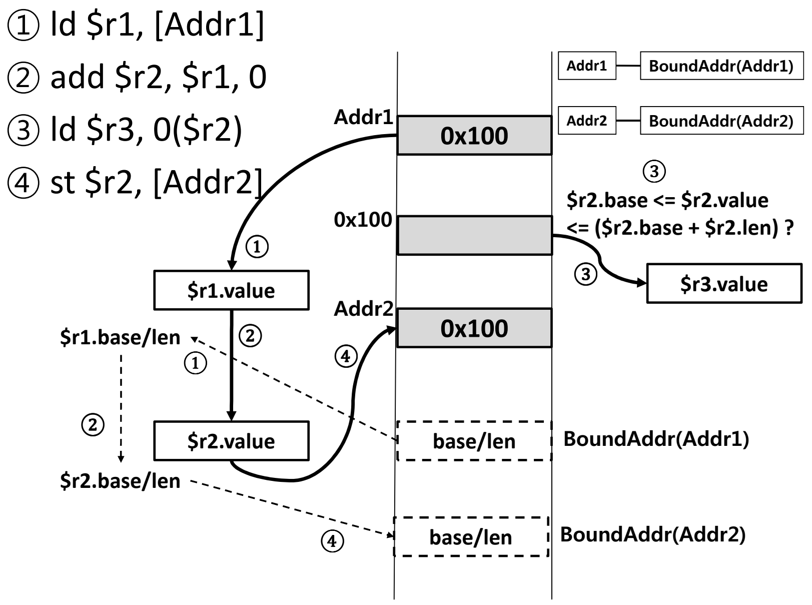


Fig. 3.1.: Bounds checking in HardBound

consider memory hierarchies and interconnects of CMP systems, which may cause significant performance overheads of data accesses for bounds checking.

3.3 Architectural Support for Efficient Bounds Checking

In this section, we propose two schemes (bounds information sharing and Smart Tagging) for fast and efficient bounds checking with hardware support. First, we explain the basic mechanism of an architectural support for bounds checking, which was introduced in HardBound [34].

All registers that can contain memory address values are expanded to have extra information for bounds checking. Table 3.1 describes the extra fields of a register to handle bounds information when a pointer is stored in the register. Figure 3.1 illus-

Table 3.1: Expanding a register for bounds checking

Extension	Description
`$r1.value`	Actual value in `$r1`
`$r1.base`	Starting address of a memory object pointed by `$r1`
`$r1.len`	Size of a memory object pointed by `$r1`

trates a simple example of handling bounds information when loading, propagating and storing a pointer. First, a pointer value 0x100 stored in Addr1 is loaded to a register `$r1`. The associated bounds information located in BoundsAddr(Addr1) is loaded to `$r1.base` and `$r1.len`. Now `$r1.value` has the address value 0x100. When `$r1` is copied to `$r2`, the bounds information stored in `$r1.base/len` is also copied to `$r2.base/len`, respectively. When the value in the memory address `$r2.value` (0x100) is loaded to `$r3`, we must make sure that `$r2.value` lies within the bounds, between `$r2.base` and `$r2.base + $r2.len`. Finally, the pointer value 0x100 in `$r2.value` is stored to the memory address Addr2. Along with it, `$r2.base/len` are also stored to the address BoundsAddr(Addr2). Note that we omit the bounds checking for Addr1 and Addr2 to simplify the description of the example, even though they must be done actually.

3.3.1 Bounds Information Sharing To Avoid Redundant Storing

The pointers in Addr1 and Addr2 in Figure 3.1 are associated with the bounds information in BoundsAddr(Addr1) and BoundsAddr(Addr2), respectively. However, note that when a pointer is copied to another pointer, they both have the same bounds information. This situation occurs frequently when a number of pointers point to the same memory object or a pointer value keeps being passed as a parameter in nested function calls. Therefore, memory overheads can be reduced if we allow pointers to share the bounds information.

Figure 3.2 describes how the bounds information can be shared when a pointer is copied to another. In order to keep track of the location of bounds information when a pointer is propagated through registers, we expand registers to have another extra *bounds* field containing the address of bounds information associated with a pointer in the register. When a pointer in Addr1 is loaded to `$r1.value`, the address BoundsAddr(Addr1) is loaded to `$r1.bounds` along with `$r1.base/len`. Also when `$r1` is copied to `$r2`, `$r1.bounds` is also copied to `$r2.bounds` together with `$r1.base/len`. When `$r2.value` is stored to the address Addr2, `$r2.base/len` do not have to be stored since they are the same as the ones in BoundsAddr(Addr1). Instead, we associate Addr2 with the address in `$r2.bounds` that is equal to BoundsAddr(Addr1) through updating mapping information between a pointer address and its bounds information address. Now the pointers in Addr1 and Addr2 share the same bounds information in BoundsAddr(Addr1).

3.3.2 Reducing Overheads Using Smart Tagging

To avoid performing bounds checking every time when a pointer is dereferenced, we can optimize the process further. First, we do not need to manage bounds in-

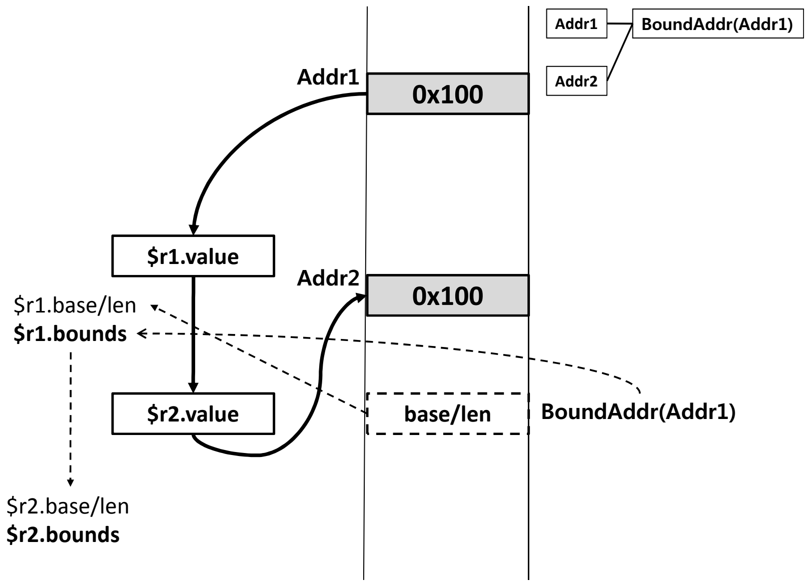


Fig. 3.2.: Sharing bounds information

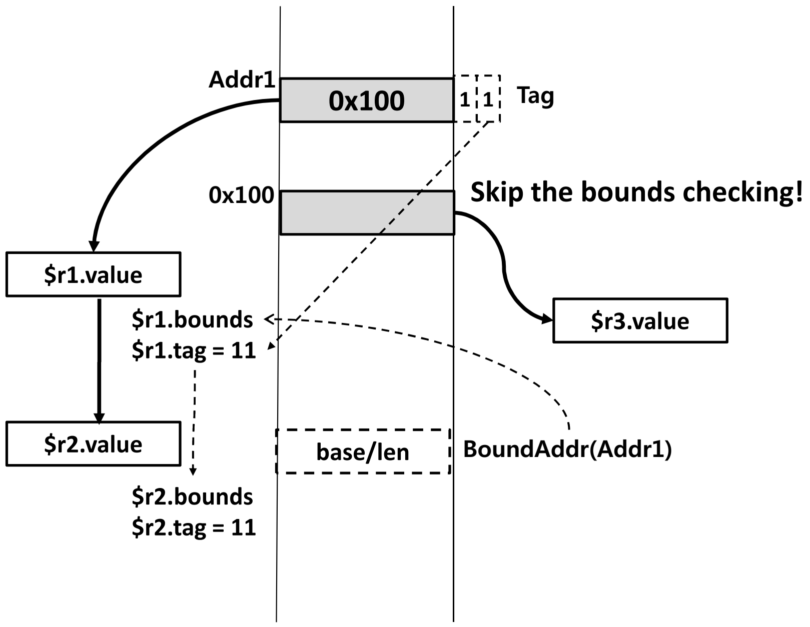


Fig. 3.3.: Skipping bounds checking with Smart Tagging

formation when handling non-pointer values in the system. And more importantly, when a pointer is initialized, it is associated with a memory object for the first time.

The pointer is safe at this moment because it points to the object correctly. Also, a pointer is guaranteed to be safe after passing the bounds checking until it is updated later. To perform bounds checking more effectively based on these observations, we use 2-bit tag information per 4-byte memory block in the address space, assuming 32-bit ISA. We need a 2-bit tag for each 4-byte block, since the pointer size is 4 bytes in 32-bit machines. The first bit in the tag represents whether the corresponding block has a pointer or a non-pointer; it is set if the value is a pointer and cleared otherwise. The second bit indicates whether the pointer value stored in the corresponding block is safe or not. It is set if the pointer is safe, when initialized or after passing the bounds checking. It is cleared if the pointer is not guaranteed to be safe, such as after a pointer update. Each register that can contain a pointer is also expanded to have the tag information called `$r1.tag`. The space overhead of the tag information in a 4GB memory address space is 256MB, occupying around 6% of the total space. Note that allocating at least one bit per 4-byte word is mandatory in 32-bit machines for bounds checking with hardware support to indicate whether each word is a pointer or not. Smart Tagging uses additional one bit per word to provide a more efficient bounds checking mechanism, which is not a big overhead compared to HardBound that uses up to four bits for each tag.

Figure 3.3 illustrates how the tag works in the previous example shown in Figure 3.1. When the value stored in Addr1 is loaded to `$r1.value`, the system first checks up the tag for Addr1. Here we assume that both the two bits are set, meaning that the value is a pointer and it is safe. Originally the bounds information should be loaded to `$r1.base/len` if the first bit is set. With Smart Tagging, however, But it is deferred until the pointer becomes stale, regardless of the second bit in the tag. The tag is loaded to `$r1.tag` in order to keep track of the safety condition of the pointer. BoundsAddr(Addr1) should be also loaded to `$r1.bounds` because the system must be able to keep track of the location of bounds information in preparation for the time when it needs to access the bounds information. When `$r1.value` is copied to `$r2.value`, `$r1.tag` and `$r1.bounds` are also propagated along with the value. Since the pointer value does not change between the source and the destination registers, `$r2.tag` remains the same as `$r1.tag`. When the value is loaded to `$r3.value` through dereferencing the pointer in `$r2.value`, we can skip the bounds checking as well as loading the bounds information because the pointer is already guaranteed to be safe by looking at the tag in `$r2.tag` that is still 11. When `$r2.value` becomes different from `$r1.value`, `$r2.tag` must be updated to 10 to indicate that the pointer is stale and needs bounds checking. Then, the bounds information in `$r2.bounds` must be loaded when `$r2.value` is dereferenced.

3.3.3 Implementation Issues

Caching Frequently Accessed Addresses of Bounds Information. For faster address translation between pointers and bounds information addresses, each core can be equipped with a bounds TLB that caches frequently accessed mapping information. Using a bounds TLB becomes greatly attractive as workloads show a higher temporal locality of bounds information.

Storing Tag Information. Unlike the bounds information shared by a number of pointers, the tag information is private to each pointer. Thus, even if multiple pointers share the bounds information, the tags for those pointers cannot be shared. Therefore, tags are to be stored in a normal L1 data cache instead of an L1 bounds cache when accessing the bounds information using the BCache architecture that will be explained in more detail in the next section.

Extracting Bounds Information from a Program. For globally declared variables or objects, bounds information can be extracted at runtime by looking up the symbol tables of executable files or libraries [50]. Also, the bounds information of dynamically allocated objects can be obtained by hacking up the system library, e.g. `glibc` for Linux, that implements `malloc()/calloc()/realloc()` interfaces. However, it is not possible to extract bounds information of local objects declared in a process stack without the help of source-level analysis. Explaining how to make analysis of local variables in a program source code is beyond the scope of our study. But it can be done easily using a source code analyzer or a compiler such as [51]. The program source code can be annotated when a local object is newly assigned to a pointer. Every time when new bounds information is obtained at runtime, the system dynamically generates a special instruction that is decoded to a generic store instruction for storing bounds information in an appropriate memory address. Note

Table 3.2: Overheads of special instructions

Benchmarks	Total bounds accesses	Special instructions	Percentage
blackscholes	417898	20	0.005%
dedup	131897	14154	10.731%
fluidanimate	15090315	405908	2.689%
streamcluster	311918	46702	14.972%
swaptions	313891	20227	6.443%
bodytrack	4681902	22403	0.478%
canneal	1295954	593125	45.767%
ferret	3898688	5994	0.153%
freqmine	3689589	89278	2.419%
Average			9.296%

that the source code modification is inevitable to extract bounds information of local objects.

3.3.4 System Overheads

Even if the bounds checking with an architectural support is fast and efficient, we need to clarify the cost of implementation precisely due to the limited resource budget of CMP systems. In this section, we measure the overheads of the proposed schemes in terms of both hardware and software.

For hardware, we obtain delay and area overheads from CACTI 5.3 [52] and HSPICE analysis in 32nm technology. A register file must be expanded to store metadata for bounds checking. The access time of a regular 128-entry integer register file is measured to be approximately 0.204ns. When the register file is expanded to have extra information for bounds checking, the access time becomes 0.222ns, showing around 9% of increase. Even this increased latency can be fit in 4GHz clock cycle time. Area overheads of the register file increases from 0.008mm^2 to 0.014mm^2, which are quite marginal compared to those of caches or interconnects in CMP systems. Also, an additional integer ALU exclusively used for bounds checking should be adopted as well in order to prevent structural hazards. Access latency of a 32-bit integer ALU is measured to be 0.048ns, and the area overheads correspond to 8.203E-6mm^2.

In the previous section, we state that newly extracted bounds information is converted to a special instruction that is eventually decoded to a store instruction. Note that this special instruction is generated only when bounds information appears in the system for the first time, and already obtained bounds information is automatically handled by hardware. Therefore, the number of special instructions generated takes up only a small portion of the total number of accesses to the bounds information in the system. Table 3.2 shows the ratio of special instructions to the bounds information accesses in PARSEC benchmark suite. The ratio is approximately 9.3% in average.

3.4 Managing Bounds Information in CMPs

In this section, we explain how to manage bounds information that may be shared by multiple threads in CMP systems. We propose a new cache architecture, which is called Bounds Cache (BCache). BCache allows duplicated copies of bounds information in L2 cache for fast accesses from threads running on multiple cores.

3.4.1 Chip-Multiprocessor Architecture

A CMP integrates more than one processor core in a single chip. In a CMP, a number of communication nodes that can be cores or caches are connected through buses or interconnects. As the system size grows, Network-on-Chip (NoC) is becoming widely adopted to provide scalability, where a router is connected to a communication node or another router through wires and transmits data. Wormhole switching is commonly adopted in CMP systems to reduce the buffer space overhead, and multiple virtual channels are deployed per each physical channel to minimize Head-of-line (HOL) blocking. A network topology determines how to connect processing elements and routers, such as a mesh or a fat tree. The CMP architecture assumed in this study is a tiled CMP in which tiles are connected via a mesh network. Each tile consists of a core with private L1 caches and a shared L2 cache bank including directory information. Figure 3.4 shows a tiled 16-core CMP architecture.

3.4.2 BCache Architecture in CMP

Figure 3.5 describes the overall BCache architecture for a 16-core CMP system. As seen in Figure 3.5a, additional L1 bounds cache (L1B) is used along with the existing L1 instruction/data caches (L1I/L1D) to manage and access bounds infor-

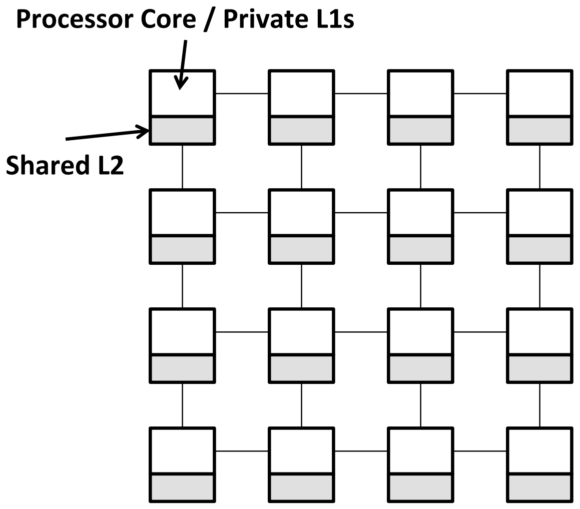
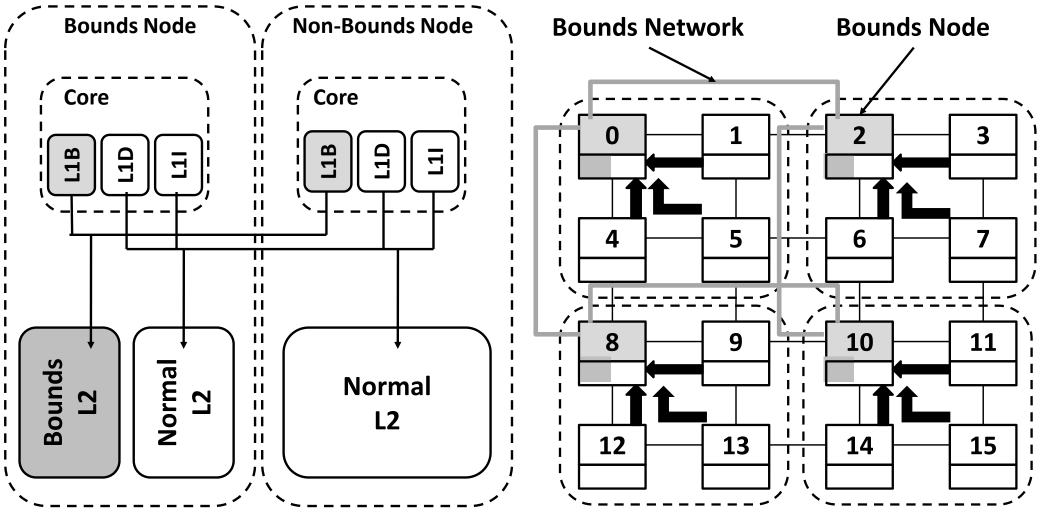


Fig. 3.4.: A tiled 16-core CMP architecture



(a) Nodes (b) Memory and interconnects

Fig. 3.5.: BCache architecture

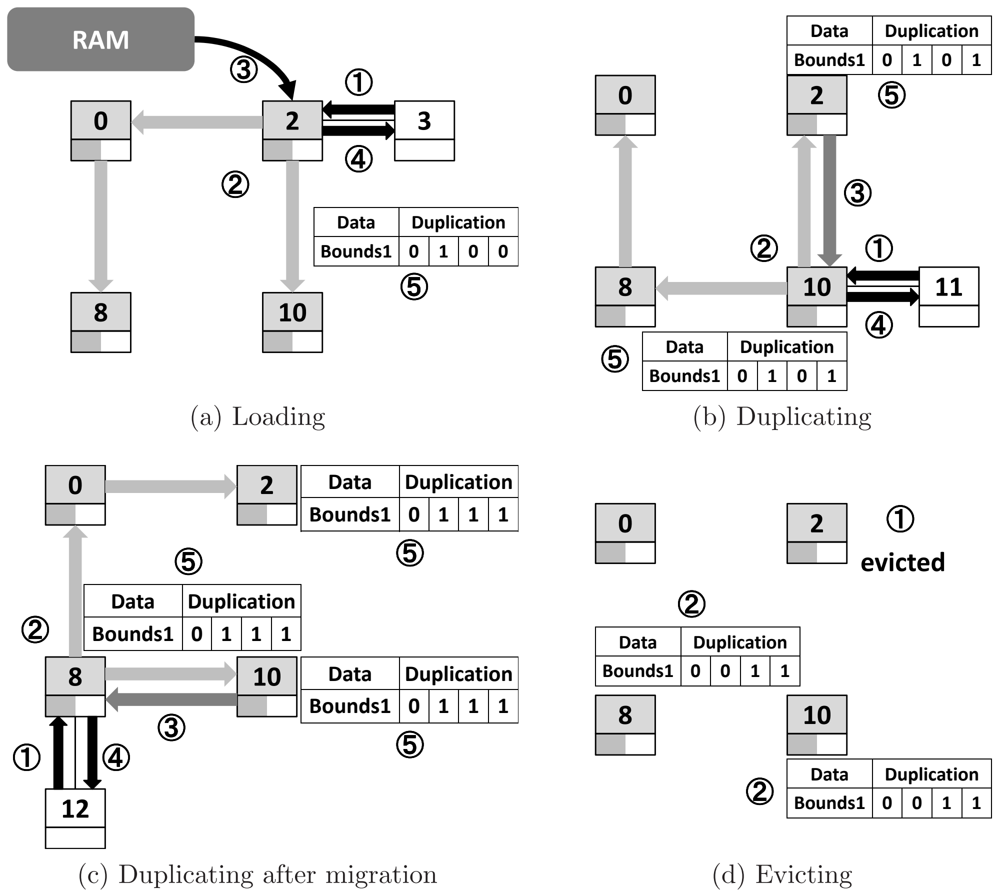
mation fast and efficiently. When bounds information is located in the L2 cache of the BCache architecture, only some specific nodes can have bounds information in their L2 caches, not allowing other nodes to store it, which is different from the normal shared L2 cache that allows bounds information to be placed in any node depending on the address. Four out of sixteen nodes, 0, 2, 8 and 10 in Figure 3.5b are chosen and they are called *bounds nodes*. Bounds nodes have a small part of their L2 caches dedicated to store bounds information and use the other part of L2 for normal data. Other non-bounds nodes have all their L2s only for regular data. Unlike the other L1 instruction/data caches, the L1 bounds cache is write-through, thus newly written bounds data in the L1 bounds cache is immediately written to the L2 bounds cache of a bounds node as well. It enables fast access to new bounds information from other nodes. For fast transmission of bounds information between bounds nodes, a separate interconnection network is adopted only for bounds data, and we call it a *bounds network* hereafter. The bounds network connects all four bounds nodes in a mesh style as depicted by gray lines in Figure 3.5b and allows communication between bounds nodes within two hops. The normal mesh network is used for other communication including normal data traffic as well as the communication between bounds and non-bounds nodes.

The most important difference between BCache and normal shared L2 cache architectures is that each bounds node can have its own duplicated copy of the same bounds information in BCache, whereas normal shared L2 cache allows only one copy in the system. To keep track of the duplication information, each cache block in L2 of BCache has extra 4-bit information indicating whether each of four bounds nodes has the same information block in its L2. All bounds nodes having duplicated copies of a bounds information block have the same duplication information for that block.

Whenever the bounds information block is duplicated or evicted in/from a bounds node, all bounds nodes sharing the block updates the duplication information. In order to make a fast access to the bounds nodes from non-bounds nodes, one bounds node and three of its neighboring non-bounds nodes are grouped together and the group is called a *bounds cluster* as shown in Figure 3.5b. When a miss occurs in L1 bounds cache of a node, a request message is first sent to the corresponding bounds node in its bounds cluster. Note that the bounds node is located close to other nodes in the cluster, which minimizes the latency of a first miss request message and that of a data message from the bounds node to the requestor. In the normal shared L2 architecture, the distance between the requestor and the owner that provides the data block may be quite far. If the bounds node has the bounds information when the bounds node receives the request message, it immediately replies to the requestor. Otherwise, it communicates with other three bounds nodes to get the data from them. All four bounds nodes efficiently share and transfer bounds information among them, which will be explained in more detail in Section 3.4.3.

3.4.3 Handling Bounds Information in BCache

The BCache architecture enables duplicated copies of a bounds information block in the last-level shared cache of a CMP system, which is different from a generic shared cache in CMP systems. Therefore, BCache cannot be used in conjunction with regular cache coherence protocols for CMPs. In this section, we introduce



(a) Loading (b) Duplicating

(c) Duplicating after migration (d) Evicting

Fig. 3.6.: Managing bounds information in BCache

a new coherence mechanism for the BCache architecture that handles duplicated bounds information effectively. Figure 3.6 illustrates how the bounds information can be managed by the BCache architecture in various situations.

Loading Bounds Information. Suppose that a cache miss occurs in the L1 bounds cache of node 3 as shown in Figure 3.6a. First, node 3 sends a request message to the bounds node in the cluster, node 2 in this example. If node 2 has the requested bounds information in its L2, it provides the information to the requestor. Otherwise, it requests other three bounds nodes (nodes 0, 8 and 10) to figure out whether any of them has the information through the bounds network [1]. If the information is found in one of the other three bounds nodes, it is transferred to node 2. Otherwise, it can be fetched from an off-chip memory. Finally, node 2 has the requested information and sends it to the requestor, node 3, using the normal mesh network. As explained in the previous section, each cache block of L2 bounds caches in bounds nodes maintains the 4-bit duplication information to keep track of the duplication status of bounds data. Thus, node 2 updates the duplication information for the current data block. If any other node in the cluster such as node 7 requests the same data later on, the node 2 can immediately provides the data from its own L2.

Loading the Same Bounds Information in a Node of a Different Cluster. Figure 3.6b shows how the bounds information previously requested by node 3 is provided to node 11 in a different cluster. Node 11 first sends a request message to the

[1]If the L2 of a bounds node does not have the data, the corresponding duplication information is not available in that node as well. So node 2 has no information of which bounds node has the data.

bounds node in the cluster, node 10, as before. If node 10 does not have the bounds information, it requests other three bounds nodes to search for the information. Since the bounds node 2 has the bounds information, it sends the information block to node 10 through the bounds network. After receiving the bounds information block from node 2, node 10 sends the information to the requesting node 11. Now the bounds nodes 2 and 10 have duplicated copies of the same bounds information in their L2. Here both the nodes 2 and 10 have knowledge of the bounds data block duplicated in those two nodes.

Loading the Bounds Information After Thread Migration. Threads running in the system keep migrating to different processor cores based on the OS scheduler's policy. Then the same bounds information may be requested from a new node after the migration. As shown in Figure 3.6c, if the thread running on node 3 is migrated to node 12, node 12 can request the bounds information that was originally used by node 3. Getting the bounds information and updating the duplication information can be done as explained above. Note that node 12 can make a fast access to the bounds information with the help of the BCache architecture, even though nodes 3 and 12 are located far from each other.

Evicting the Bounds Information from a Bounds Node. If new bounds information is written to the bounds node's L2 cache whose corresponding cache set is already full, one bounds cache block must be chosen for eviction based on the LRU replacement policy. In Figure 3.6d, when a bounds information block is duplicated in the bounds nodes 2, 8 and 10 and the one in node 2 is evicted, node 2 sends the update message to the other two bounds nodes 8 and 10 to update the duplication information. Once nodes 8 and 10 receive the update message, they delete node 2 from the duplication information. If the evicted block is the only copy and it is not stored in the off-chip memory yet, it is written back to the memory after eviction.

Modifying the Existing Bounds Information. The bounds information can be modified at runtime by `realloc()` and so on. If it is modified, all the previous copies of that bounds information must be invalidated. Assume that the `realloc()` is called in node 3 to modify the bounds information. Right after new bounds information is stored in the L1 bounds cache of node 3, it is immediately sent to the bounds node 2 for update since the L1 bounds cache is write-through as explained in Section 3.4.2. Then the bounds node 2 broadcasts the new bounds information to all other three bounds nodes to invalidate the old information through the bounds network. Once each bounds node receives the invalidation message, it updates the duplication information and broadcasts the message again to its neighboring nodes in the cluster through the normal mesh network for invalidation. After invalidation, the new bounds information is located only in the bounds node 2, and getting the new information from other nodes can be done in the same way as explained above. This broadcast-based invalidation procedure may increase the overheads compared to the normal cache coherence protocol, since an invalidation message must be broadcast to all nodes every time a write occurs. But usually updating bounds information is very rare in a program and the invalidation is off the critical path, so it will not make a big impact on the performance degradation.

3.4.4 Scalable Design of BCache Architecture

When a CMP system scales up to have more processor cores in a chip, each bounds node must be properly located in a cluster for optimal performance. Thus, for each node, we calculate the average distance to reach all nodes in a cluster. Suppose that an (a x b) bounds cluster is placed on an imaginary X-Y graph, where each node corresponds to one of the coordinates on the graph between $(0, 0)$ and $(a - 1, b - 1)$. Assuming that C is the set of all nodes in a cluster, average distance from an arbitrary node $p = (x, y) \in C$ to any other node (x_1, y_1) in C can be obtained using the following equation 3.1.

$$AvgDist_p = \frac{\sum_{(x_1,y_1)\in C}(|x - x_1| + |y - y_1|)}{a \times b} \tag{3.1}$$

We can generalize this formula to a mixed radix $k_0 \times k_1 \times \cdots \times k_{n-1}$ n-mesh network. Here the distance between $p = (p_0, p_1, \cdots, p_{n-1})$ and $q = (q_0, q_1, \cdots, q_{n-1})$, where $p, q \in C$, is defined as the following equation 3.2.

$$Dist(p, q) = \sum_{i=0}^{n-1} |p_i - q_i| \tag{3.2}$$

The average distance from an arbitrary node $p = (p_0, p_1, \cdots, p_{n-1})$ to all other nodes in the cluster is defined as the following equation 3.3.

$$AvgDist_p = \frac{\sum_{q \in C}(Dist(p, q))}{\prod_{i=0}^{n-1} k_i} \tag{3.3}$$

Using this formula, we choose the proper location of a bounds node that minimizes average distance. Figure 3.7 illustrates the BCache architectures deployed in 4x4, 8x4, 8x8 and 16x8 2-D mesh CMP systems.

In the case of the concentrated mesh (CMesh) topology [53] in which routers are connected in a mesh style and each router services four nodes, we can treat those nodes sharing a router as a group. Then, the topology becomes a normal mesh connecting those groups. Here we can select the group with the shortest average distance using the equation 3.3. Then we can choose any node in that group as a bounds node since all nodes have the same distance from the router.

3.4.5 Design Alternatives for Large-Scale BCache Architecture

As the system size increases in a large scale, a distance between a bounds node and other nodes in a bounds cluster also increases in 2D mesh-style topologies. To overcome network overheads, a three-dimensional (3D) die-stacked architecture might be adopted to shorten the latency in a more efficient way. In the 3D architecture, multiple silicon dies are stacked together and they communicate each other through

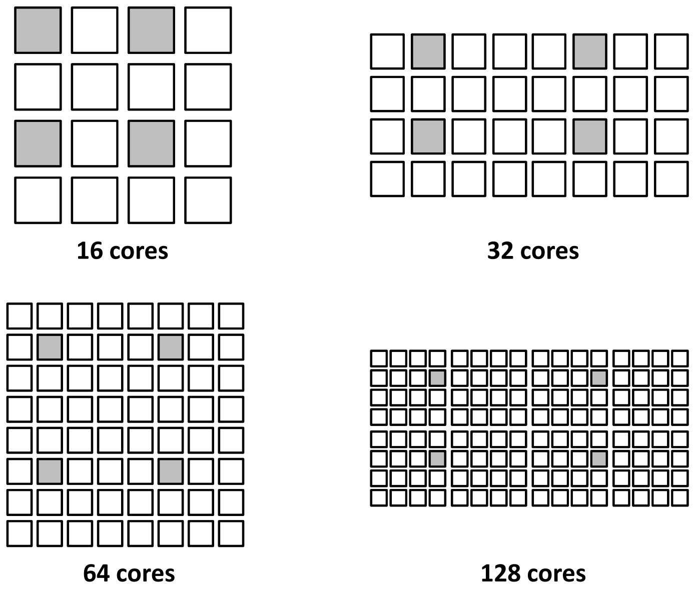
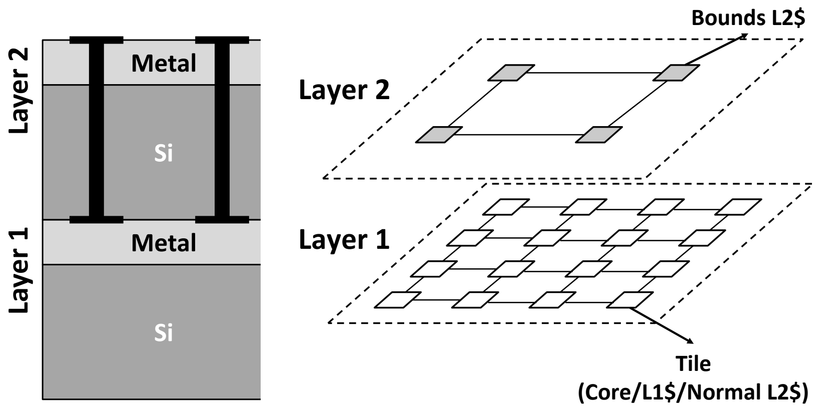


Fig. 3.7.: Scalable BCache design in CMP systems



(a) Face-to-Back (F2B) bonding

(b) 3D design of BCache architecture

Fig. 3.8.: Design alternative using 3D stacking

vertical interconnects. Figure 3.8a describes a 3D architecture with two layers connected with Face-to-Back (F2B) bonding using Through Silicon Vias (TSVs) .

The BCache architecture can be designed using two different silicon layers as shown in Figure 3.8b. The first layer contains nodes with cores, L1 caches and normal

L2 caches connected through a normal mesh network. Bounds nodes connected through the bounds network are located in the second layer. Then, the access delay from the non-bounds nodes to the bounds node reduces dramatically. The amount of performance enhancement is proportional to the number of TSVs used, but too many vertical interconnects might take up much space and compete with other devices for the area budget. Here we assume that each bounds cluster has four TSVs, each of which is 128 bits wide. Each TSV handles the communication between the upper-level bounds node and one quadrant of cluster nodes in the lower layer. Among the nodes in each quadrant, the one with the shortest average distance to others is connected to the TSV. The location of that node can be determined using the method in Section 3.4.4. Each bounds node in the upper layer is connected to a 5x5 NoC router equipped with four channels connected to four TSVs and another channel for injection and ejection.

As the number of TSVs per bounds cluster increases in a large-scale CMP system, the router connected to the bounds node in the upper layer might become significantly complicated. One solution is that we can make all vertical connection points in the upper layer connected to a high-speed media such as an optical shared bus in the manner of a Hamiltonian path [54]. The high-speed shared bus also helps to communicate between bounds nodes when transmitting bounds data or updating the sharing information through broadcasting.

3.4.6 Using BCache for General Purposes

One may consider BCache is too expensive to be used exclusively for bounds checking. Here we prove that BCache can be used more generally by showing how to use BCache for regular data or make BCache operate as a generic coherent cache.

In addition to bounds information, BCache can be used to store regular data accessed by normal load and store instructions. Since BCache is not efficient to handle heavily overwritten data due to the invalidation overheads, read-only data would be a good candidate for BCache. Storing read-only data in BCache can be done in the same way as the bounds information that is explained in Section 3.3.3.

Also, BCache can be reconfigured as a part of the normal cache hierarchy when the bounds checking is not necessary. Bounds caches in L1 and L2 can work as a part of normal L1 data or L2 caches by reconfiguring the L1 bounds cache from write-through to write-back and assigning a portion of physical memory address space to BCache exclusively. If the memory address belonging to that address range is accessed by a load or store instruction, L1 bounds cache is looked up instead of normal L1 data cache. A cache miss in an L1 bounds cache is handled by one of L2 bounds cache banks depending on the address. Note that L2 bounds caches do not allow data duplication in this case, so each cache block is stored in only one L2 bounds cache bank. To maintain cache coherence, each cache line in L2 bounds caches has directory information in addition to the existing 4-bit sharing information. As in the bounds checking, packet traversal between bounds nodes is done via the

Table 3.3: System parameters

Simulators	Parameters	Values
Simics/ FeS_2	CPU	16 Intel x86 processors
	Cache block size	64B
	Memory	1GB, 300 cycle access time
	Directory access time	80 cycles
	OS/Kernel	Fedora Core 5 (x86) / Linux 2.6.15
CMP CacheSim	L1 cache	4-way, 1KB(L1B/L1D), 2 cycles HardBound: Write-back L1B BCache: Write-through L1B
	L2 cache	4-way, 16MB, 6 cycles HardBound: 1MB x 16 BCache: 1MB x 12 (non-bounds nodes) 512KB x 4 (bounds nodes/bounds data) 512KB x 4 (bounds nodes/regular data)
	Cache block size	64B
	Memory	1GB, 300 cycle access time
	Router	Fixed 5-cycle pipelined router
	Flit size	16B
	Network topology	4x4/8x4/8x8/16x8/16x16 mesh

bounds network. Also, the traversal between bounds and non-bounds nodes is done via a normal mesh network. The configuration of L1 bounds caches and the memory address range can be supported by system BIOS.

3.5 Performance Evaluation

3.5.1 Simulation Framework

We measure the performance of the proposed schemes using Simics full-system simulator [25]. In order to simulate the bounds checking mechanism explained in the previous section, we also use a Full-system Execution-driven Simulator for x86 (FeS_2)

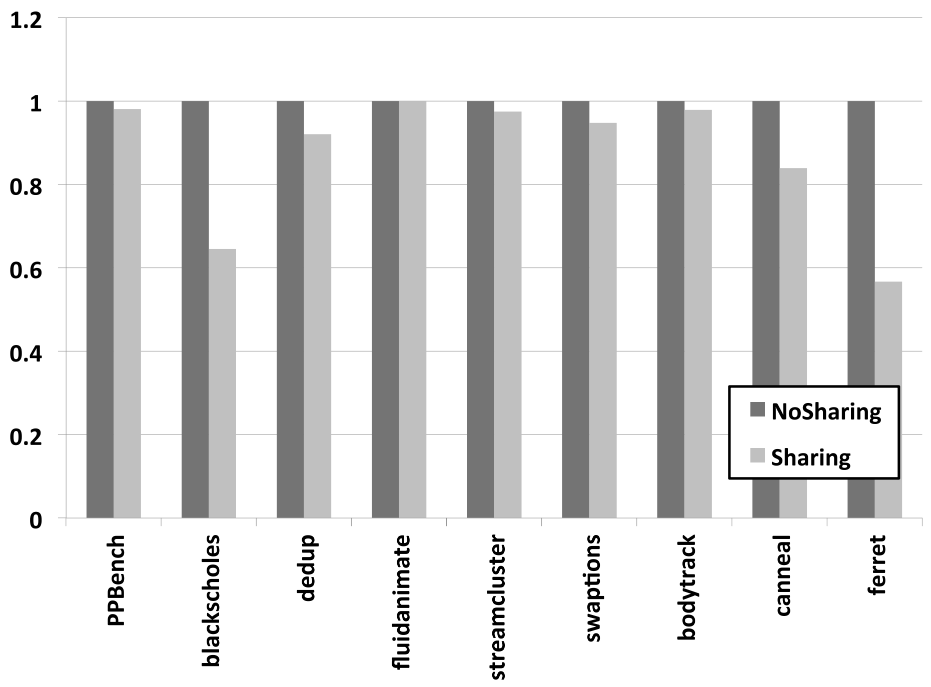


Fig. 3.9.: Effect of sharing bounds information

Table 3.4: Number of bounds checking

Benchmarks	Total	Skipped	Percentage
PPBench	296907	295954	99.679024%
blackscholes	3247416	3247395	99.996921%
dedup	53458	46598	87.167496%
fluidanimate	812156	809873	99.718896%
streamcluster	522627	519172	99.338917%
swaptions	11840543	11834754	99.951109%
bodytrack	4659151	4596920	98.664327%
canneal	806383	802427	99.509414%
ferret	3884803	3766475	96.954080%
freqmine	3616691	3441109	95.145231%
Average			97.612785%

that models out-of-order x86 processor cores including a decoder used to convert an x86 instruction to RISC-style micro-ops. FeS_2 integrates PTLsim [55] decoder and Ruby memory model in GEMS [26]. To measure the BCache architecture performance, we capture all memory reference traces including bounds information using Simics and FeS_2. Then those traces are fed to a cycle-accurate CMP cache simulator that models three different cache architectures: HardBound, BCache, and BCache with 3D stacking. Note that HardBound is designed for uniprocessor systems, so it does not assume a specific cache coherence protocol. Hence we assume a general directory-based MSI coherence protocol for HardBound, while BCache uses its own protocol explained in Section 3.4.3. CACTI 5.3 [52] and ORION 2.0 [56] are used to measure energy consumption for caches and interconnects, respectively. All energy results are obtained on the assumption of 32nm technology.

Table 3.3 shows system parameters used in the simulation. To make a fair comparison, the total size of L2 bounds caches in BCache is set to the same as that of L2 cache in HardBound. Also, we assume that HardBound stores the bounds information in L1 data caches and the tag information in L1 bounds caches. BCache stores the tag information in L1 data caches, as explained in Section 3.3.3.

We use PARSEC [29] benchmarks with `simsmall` input sets for parallel workloads. Also we make our own parallel benchmark called *ParallelPointerBench* to measure the performance with more pointer-intensive workloads. ParallelPointer-Bench spawns threads as many as the number of cores and generates total 2,000 global and thread-specific pointers. Each thread accesses pointers 2,000 times at random. All benchmark source codes have been modified to extract read-only data and bounds information for pointers by adding a simple notation. Note that we evaluate only the benchmark code, not libraries and OS kernel.

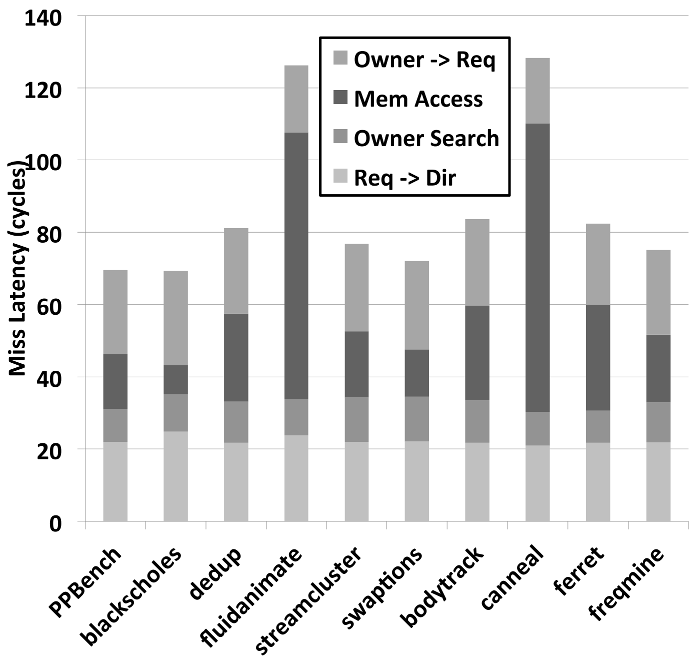
In order to measure the BCache architecture scalability, we need memory reference traces for systems with more than 16 processor cores. However, we could not use Simics to generate traces for large-scale systems because of prohibitive simulation time. Instead, we make a trace generator that is able to create traces for an arbitrary number of cores. The generator creates traces simulating workloads generating and accessing pointers randomly for 16, 32, 64, 128 and 256-core systems. Based on the behavior of real parallel benchmarks, we set the ratio of writes to read operations to be 3% in each trace.

3.5.2 Simulation Results

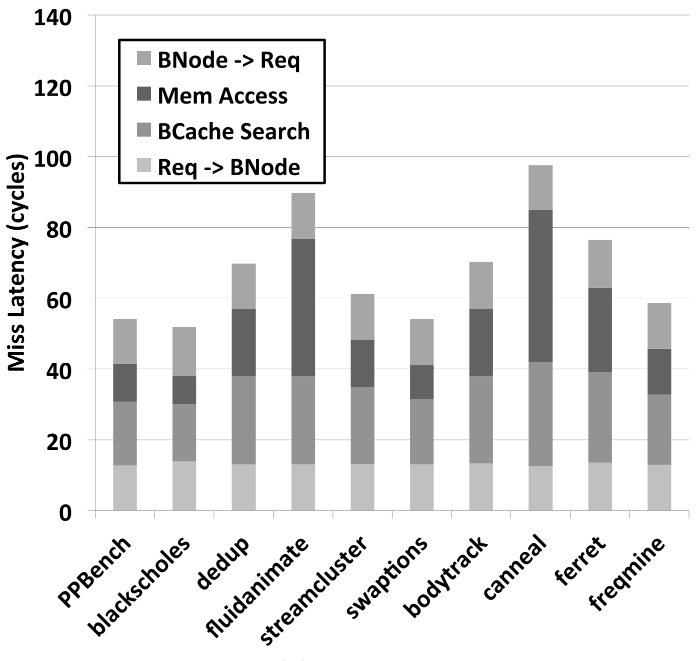
First, we clarify how much the space overhead can be reduced through the sharing of bounds information among multiple pointers. Figure 3.9 shows the amount of an address space allocated to the bounds information. The first bar in each column represents the result when bounds information is not shared. The second bar depicts the result of shared bounds information normalized to the first one. Figure 3.9 shows that 15% of space overheads can be reduced on average when pointers to the same object share the bounds information.

Also, we show the amount of skipped bounds checking through Smart Tagging in detail. Table 3.4 shows the ratio of skipped bounds checking with Smart Tagging out of the total bounds checking. It shows approximately 98% of bounds checking can be skipped on average.

We clarify how much the BCache architecture is beneficial to fast access to the bounds information in CMP systems. Figure 3.10 shows average miss latencies of loading and storing bounds information for HardBound, BCache and BCache-3D designs. In each graph, the first column *PPBench* represents ParallelPointerBench and the next columns shows PARSEC benchmark results. To compare the results in more detail, we break down the latency values into four parts for each design. In HardBound shown in Figure 3.10a, the miss latency consists of the request latency from a requestor to the corresponding directory node (**Req → Dir**), the lookup latency needed to reach the owner (**Owner Search**), the off-chip memory access latency in the case of L2 miss (**Mem Access**), and the transfer latency from the owner to the requestor (**Owner → Req**). Similarly, the miss latency of BCache in Figure 3.10b consists of the request latency from a requestor to its bounds node (**Req → BNode**), the lookup latency for fetching data among four bounds nodes and placing it in the requested bounds node (**BCache Search**), the off-chip memory latency (**Mem Access**), and the data transfer latency from the requested bounds node to the requestor (**BNode → Req**). In Figure 3.10a and 3.10b, BCache improves the miss latency by 28% on average compared to HardBound. Note that BCache



(a) HardBound



(b) BCache

Fig. 3.10.: Average miss latencies of accessing bounds information

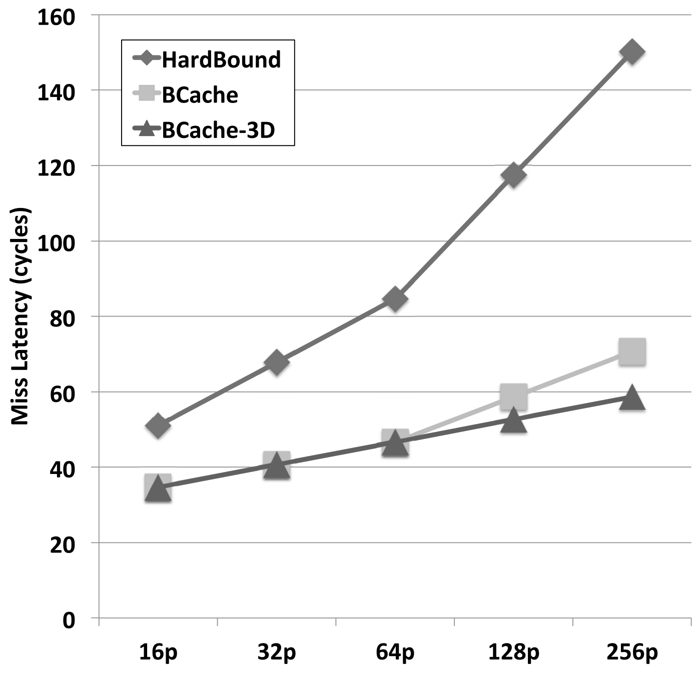


Fig. 3.11.: Scalability of BCache architecture

achieves significant performance improvement in terms of the first request and the final data transfer latencies since the bounds node providing the data is located close to the requestor. BCache reduces off-chip memory access latency because it places duplicated copies of the bounds cache block in multiple bounds nodes in a smart way, resulting in reducing the number of L2 misses in bounds nodes.

To measure the scalability of BCache in large-scale systems, we evaluate miss latencies in 16, 32, 64, 128 and 256-core CMP systems using synthetic traces obtained from the trace generator. In Figure 3.11, the performance improvement increases significantly as the system size grows, and it goes up to 156% in the case of a 256-core CMP system using BCache with 3D stacking. 3D stacking contributes to this improvement by allowing all nodes in the lower layer to reach its bounds node with shorter delays through the TSVs. Moreover, it is found that even the results of BCache without 3D stacking shows quite a good scalability. Here we fix total number of accesses to the bounds information for all different system sizes. Thus, the injection rate per node decreases as the system size grows. Therefore, the results in this experiment do not include any delay caused by contention as the network size grows. If the injection rate per node is kept constant as the system scales up, we may observe some congestion delay in communication using bounds network. For this study, we focus on the delay caused by topologies.

We also measure the average miss latencies for regular read-only data. We do not show the results from BCache with 3D stacking since BCache with 3D stacking has similar results with BCache under a 16-core system in terms of the average miss latency as shown in Figure 3.10. In Figure 3.12, BCache outperforms HardBound again by 49% on average. BCache is designed to handle read-only data efficiently as well as bounds information, while HardBound assumes general memory and cache coherence models with no optimization. Therefore, HardBound does not show performance improvement for read-only data.

Now, we investigate how BCache and Smart Tagging affect the overall system performance together. We use Simics and FeS_2 to get the baseline performance, and apply miss latency results obtained from our cache simulator and traces that record all accesses to read-only data and bounds information as well as tag lookup operations. Figure 3.13 shows the performance of three different schemes in terms of the number of cycles per micro-op (CPμ); HardBound, BCache and BCache with Smart Tagging. We use CPμ instead of CPI as a performance metric since each x86 instruction is decoded to a number of micro-ops as explained before. Figure 3.13 shows CPμ of all memory operations executed, including results of a system that does not perform bounds checking (*Unchecked* in the graph). Using BCache with Smart Tagging improves performance by 11% on average compared to HardBound, which corresponds to 24% of the gap between HardBound and no bounds checking systems, the maximum amount of improvement possible. Note that we handle only applications, not libraries and OS kernel. Therefore, the amount of improvement will increase significantly if we perform the bounds checking for all those codes. The performance improvement of skipping bounds checking looks marginal. Since PARSEC benchmark suite is not pointer-intensive, the cache traffic of bounds information is sparse compared to that of all memory operations. A noticeable performance improvement in the pointer-intensive ParallelPointerBench supports our analysis.

Figure 3.14 describes how much energy consumption can be reduced through the BCache architecture, in both benchmarks and synthetic workloads. It shows the amount of energy consumed by caches and interconnects in HardBound and BCache accessing bounds information, and the results are normalized to those of HardBound. BCache reduces the energy consumption by 47% for caches and 61% for interconnects on average. This proves that the efficient management of bounds information in BCache significantly contributes to improving energy efficiency. Especially, the

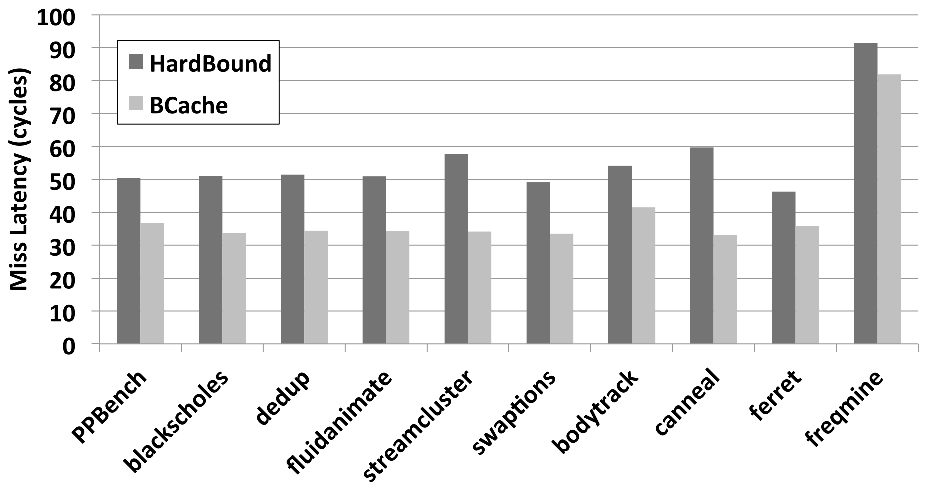


Fig. 3.12.: Average miss latencies for read-only data

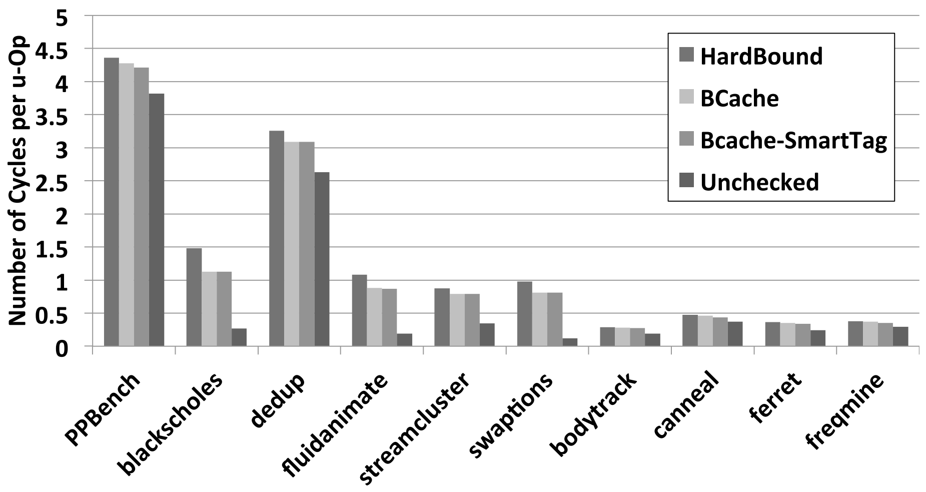


Fig. 3.13.: Performance improvement of memory accesses

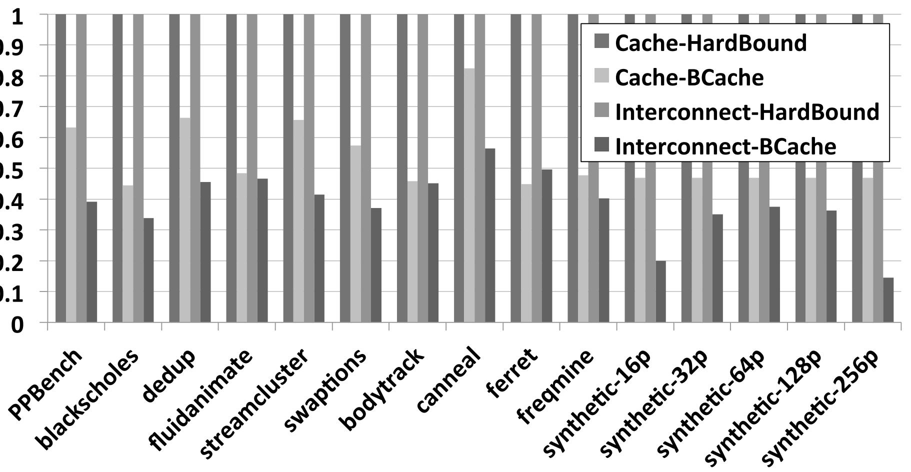


Fig. 3.14.: Energy consumption in caches and interconnects

Table 3.5: Storage overheads

Level	HardBound	BCache
L1	2KB/8KB per node (tag cache)	1KB per node (L1 bounds cache)
L2	use normal L2 cache	512KB x 4, a part of total L2 cache (L2 bounds cache)

reduced number of hops in the BCache interconnect helps to save the dynamic energy in links and NoC routers.

Finally, we compare the extra overheads of HardBound and BCache for storing bounds information in Table 3.5. The overheads of BCache are only 25% to 50% of the HarBound overheads. Therefore, BCache does not increase the overheads much, compared to HardBound.

3.6 Conclusions

In this study, we have proposed an architectural support for fast and efficient bounds checking for multi-threaded workloads in CMP systems. We reduce the space overheads through bounds information sharing and adopt Smart Tagging that enables the skipping of bounds checking for pointers already guaranteed to be safe. The BCache architecture allows fast delivery of bounds information as well as regular read-only data to a requestor node by duplicating the same data block that might be shared by threads in multiple locations. Simulation results show that the proposed schemes reduce the memory space allocated for bounds information by 15% as well as the number of bounds checking by 98% on average. Overall performance is improved by 11% on average in terms of CPμ of memory operations compared to HardBound. Also, energy efficiency also increases by 47% and 61% on average in caches and interconnects, respectively.

Our work can be explored further by investigating the address mapping mechanism of bounds information in more detail. Also we plan to examine the BCache design for other topologies such as fat tree or flattened butterfly.

4. PERFORMANCE AND POWER-EFFICIENT INPUT BUFFER DESIGN FOR ON-CHIP INTERCONNECTS

4.1 Introduction

With the continued advance of CMOS technology, the number of cores on a single chip keeps increasing at a rapid pace. And it is highly expected that many-core architectures with more than hundreds of processor cores will be commercialized in the near future. In a large-scale chip multiprocessor (CMP) system, network overheads are more dominant than computation power in determining overall system performance. While shared buses provide networking performance enough for a small number of CMP nodes, they cannot be good solutions for many-core systems due to the limitation on scalability. Accordingly, switch-based networks-on-chip (NoCs) are being adopted as an emerging design trend in many-core CMP environments. Since all components in a chip including processors, caches and interconnects must compete for limited area and power budgets, resources available for NoCs are tightly constrained compared to off-chip interconnects. Moreover, network performance becomes more significant with the increasing scale of CMP systems. Therefore, a new and innovative NoC design that can guarantee better performance with limited resources is necessary for many-core systems.

The advance of memory technology has ushered in new non-volatile memory (NVM) designs that overcome the drawbacks of existing memories such as SRAM or

DRAM. Among them, Spin-Torque Transfer Magnetic RAM (STT-MRAM) is being regarded as a promising technology for a number of advantages over the conventional RAMs. STT-MRAM is a next-generation memory that uses magnetic materials as the main information carrier. It achieves lower leakage power and higher density compared to the existing SRAM. Also, STT-MRAM shows higher endurance compared to other NVM techniques such as Phase Change Memory (PCM) or Flash, which makes STT-MRAM more attractive for on-chip memories that must tolerate much more frequent write accesses compared to off-chip memories. However, one of the biggest weaknesses of STT-MRAM is long write latency compared to SRAM. Since the fast access time of memories on a chip must be guaranteed and cannot be negotiable, the slow write operations of STT-MRAM limit its popularity, even though it shows competitive read performance. Another serious drawback of STT-MRAM is high power consumption in write operations. This issue of high power consumption in STT-MRAM must be resolved in NoCs due to the limited power budgets.

Despite these weaknesses, using STT-MRAM in the NoC design has significant merits since an on-chip router can incorporate larger input buffers compared to SRAM with the same area budget because of the higher density of STT-MRAM. Larger input buffers contribute to improving the throughput of NoC, which results in the enhancement of overall system performance. However, the aforementioned challenges must be addressed first to exploit the benefit of STT-MRAM in NoC.

Since the input buffer of an on-chip router must handle arriving flits on time, it is impossible in reality to use STT-MRAM without additional technique to hide the long write latency. Moreover, addressing the high write power issue of STT-MRAM is mandated in NoC environments.

In this study, we explore the design issues of adopting STT-MRAM in on-chip interconnects. First, by relaxing the non-volatility of STT-MRAM, the latency as well as the power consumption in write operations can be reduced at the sacrifice of the retention time [57], [58]. Based on the observation of intra-router latency of flits, we find out that the retention time needed for input buffers in NoC can be significantly shortened. We exploit the write latency reducing technique [57] in the input buffers of on-chip routers, and decrease the latency to less than 2ns that corresponds to 6 cycles in 3GHz clock frequency. Then we propose a hybrid design of input buffers combining both SRAM and STT-MRAM. By allowing each arriving flit to be stored in the SRAM buffer first and then migrated to STT-MRAM, the write latency of STT-MRAM is effectively hidden, thus increasing network throughput.

Simply migrating each flit from SRAM to STT-MRAM buffer causes significant power consumption due to the high write power of STT-MRAM, compared to existing SRAM-based input buffers. So we design a lazy migration scheme that allows the flit migration only when the network load exceeds a certain threshold, which helps to reduce the power consumption significantly. Simulation results show that the hybrid input buffers improve the network throughput by 21% in synthetic workloads and

14% in SPLASH-2 parallel benchmarks on average compared to pure SRAM-based buffers with the same area overheads. Also, the lazy migration scheme contributes to power reduction by 61% on average compared to the simple migration scheme that always migrates flits from SRAM to STT-MRAM.

The remainder of this study is organized as follows. We discuss related work in Section 4.2, followed by the performance and power model of STT-MRAM in Section 4.3. In Section 4.4, we explain the hybrid buffer design using STT-MRAM in detail. Section 4.5 presents simulation results and analysis, and finally Section 5 summarizes our work and makes conclusions.

4.2 Related Work

Since there has been no prior work using STT-MRAM in NoC design, we only summarize the relevant studies of STT-MRAM technologies as well as the application of NVM to diverse system domains such as processors and memories.

4.2.1 STT-MRAM

STT-MRAM is a next generation memory technology that takes advantage of magnetoresistance for storing data. It uses a Magnetic Tunnel Junction (MTJ), the fundamental building block, as a binary storage. An MTJ comprises a three-layered stack: two ferromagnetic layers and an MgO tunnel barrier in the middle. Among them, the fixed layer located at the bottom has a static magnetic spin, the spin of

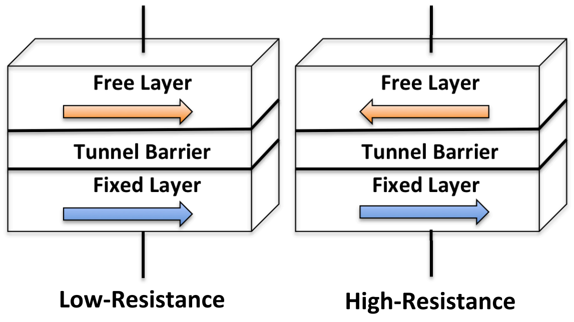


Fig. 4.1.: The two states of an MTJ module

the electrons in the free layer at the top is influenced by applying adequate current through the fixed layer to polarize the current, and the current is passed to the free layer. Depending on the current, the spin polarity of the free layer changes either parallel or anti-parallel to that of the fixed layer. The parallel indicates a zero state, and the anti-parallel a one state. Figure 4.1 depicts the two parallel and anti-parallel states of an MTJ module. A single MTJ module is coupled with a transistor to form a basic memory cell of STT-MRAM called a 1T-1MTJ cell.

4.2.2 Utilizing NVMs in Processors and Memories

Several schemes have been proposed to provide architectural support for applying NVMs to system components. Jog *et al.* [57] proposed to achieve better write performance and energy consumption of STT-MRAM-based L2 cache through adjusting data retention time of STT-MRAM. Similarly, Smullen *et al.* [58] reduced the write latencies as well as dynamic energy of STT-MRAM by lowering the retention time for designing on-chip caches. In [59], they integrated STT-MRAM into on-chip caches in a 3D CMP environment and proposed a mechanism of delaying cache accesses to busy STT-MRAM banks to hide long write latency. Prior to that, Sun *et al.* [60] stacked MRAM-based L2 caches on top of CMPs and reduced overheads through read-preemptive write buffer and hybrid cache design using both SRAM and MRAM. Guo *et al.* [61] resolved the design issues of microprocessors using STT-MRAM in detail for more power-efficient CMP systems.

PCM also has been constantly explored to replace existing SRAM or DRAM-based memory systems. Due to its lower endurance compared to SRAM or STT-MRAM, PCM is mainly adopted for off-chip memories rather than on-chip caches. Several designs of PCM-based main memory were discussed in [62], [63], [64]. In [65], adaptive write cancellation and write pausing policies were proposed to reduce energy and improve performance. Zhou *et al.* [66] suggested a new memory scheduling scheme that allows Quality-of-Service (QoS) tuning through request preemption and row buffer utilization.

4.3 Performance and Power Model of STT-MRAM

As an area model of STT-MRAM, we use ITRS 2009 projections [67] as well as the model used in [61], where a 1T-1MTJ cell size is $30F^2$ in the 32nm technology. When we assume that an SRAM cell size is approximately $146F^2$ with the same technology, one SRAM cell can be substituted by at least four STT-MRAM cells under the same area budget. Also, about 3.2ns of write latency can be achieved with $30F^2$ STT-MRAM cell size [61]. It corresponds to 10 cycles in 3GHz clock frequency, which is quite long for on-chip routers compared to SRAM that completes both read and write accesses in a single cycle. Reducing retention time from 10 years to 10ms guarantees the same write latency with one third of original write current needed [57]. Using lower current is beneficial in terms of area overheads because it facilitates to implement STT-MRAM cells with smaller transistors, which reduces actual cell area.

In this study, we slightly increase write current to reduce this write latency of STT-MRAM further. The write latency reduces from 3.2ns to 1.8ns through increasing the write current from 50μA to 75μA under $125\,^{\circ}$C of a temperature. Note that even this increased current is far less than the original current needed for 10 years of retention time, while maintaining the same STT-MRAM cell size, $30F^2$. Also, the increased current does not hurt write energy consumption since the MTJ switching time decreases accordingly [61]. As a result, the write latency decreases from 10 to 6 cycles in 3GHz clock frequency. The increased write current may hurt the performance in terms of read latency. However, we verify that the reduction of write latency from 3 to 1.8ns affects the read latency to only a small extent [58]. Therefore, we can assume that the increased read latency can still be covered by a single cycle, considering the original read delay of 122ps [61], which is far shorter than 333ps, a cycle time in 3GHz clock frequency.

The relaxed retention time of 10ms may hurt the reliability of data stored in an STT-MRAM buffer, if the retention time is shorter than the intra-router delay of a flit, defined by the time difference between arrival time at the buffer and departure time in a router. Figure 4.2 depicts maximum intra-router latency for different injection rates ranging from 0.1 to 0.7 with various SRAM buffer sizes per VC, under uniform random synthetic workloads. We observe that the latency does not go up beyond 16 cycles, and it is almost negligible compared to 10ms, which corresponds to more than 30 million cycles in 3GHz clock frequency [1]. Hence, it is confirmed that even the reduced retention time is completely enough to hold a flit in STT-MRAM buffers safely. For the read and write energy model of STT-MRAM, we conservatively adopt the same parameters from [61], 0.01pJ and 0.31pJ per bit for read and write, respectively. Note that these are based on 3.2ns of write latency, so actual write energy becomes smaller after decreasing the latency to 1.8ns.

4.4 An On-Chip Router Architecture with Hybrid Buffer Design

In this section, we describe a generic router architecture and a buffer structure in NoC and present our hybrid buffer design that maximizes the mutually complementary features of the two different memory technologies, SRAM and STT-MRAM, while minimizing the drawbacks of STT-MRAM, the long latency and high power consumption in write operations.

[1]Note that in deadlock situations, packets can stay in the network forever. In this study, we adopt deadlock-free routing algorithms, thus avoiding such situations.

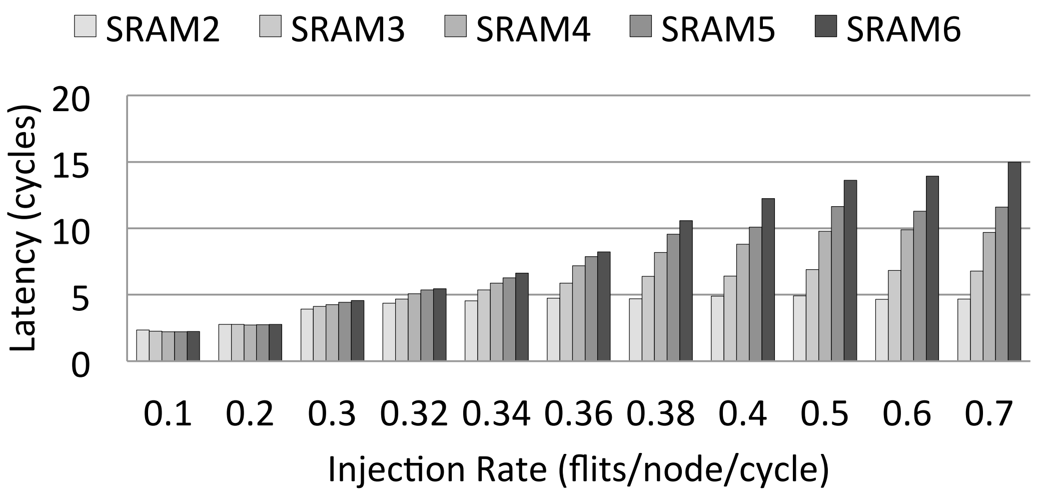


Fig. 4.2.: Maximum intra-router latency of an on-chip router (*SRAM#*: SRAM buffer size per VC)

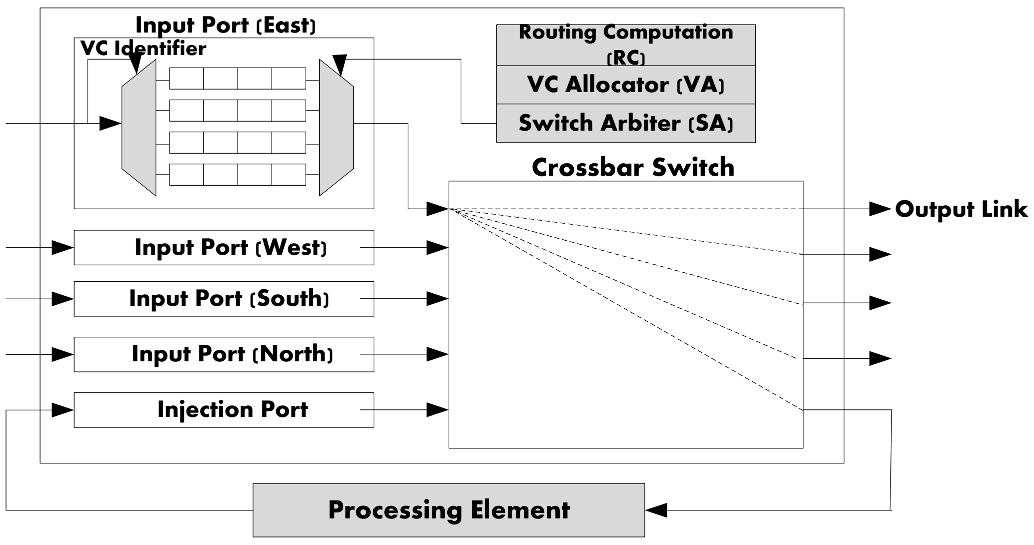


Fig. 4.3.: Generic router architecture

4.4.1 Generic Baseline Router Architecture

The generic NoC router architecture is depicted in Figure 4.3. It is based on the state-of-the-art speculative router architecture [68]. Each arriving flit goes through

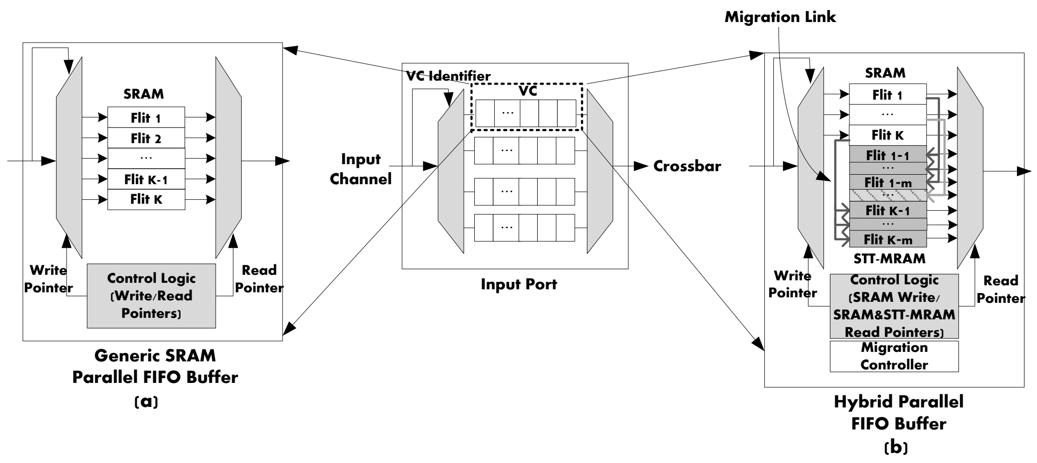


Fig. 4.4.: A generic SRAM input buffer (a) and a hybrid input buffer (b)

2 pipeline stages in the router: routing computation (RC), VC allocation (VA) and switch arbitration (SA) at the first cycle, and switch traversal (ST) at the second cycle. A lookahead routing scheme [69] is adopted, which generates routing information of the downstream router for an incoming flit prior to the buffer write, thus removing the RC stage from the critical path. Each router has multiple VCs per input port and uses flit-based wormhole switching [70]. Credit-based VC flow control [71] is adopted to provide the back-pressure from downstream to upstream routers, thus controlling flit transmission rate to prevent packet loss due to buffer overflow.

Due to the limited area and power resources and ultra-low latency requirements, on-chip routers rely on very simple buffer structure. VC-based NoC routers consist of a number of FIFO buffers per input port where each FIFO corresponds to a VC as illustrated in Figure 4.4a. Each input port has v VCs, each of which has a k-flit FIFO buffer. Current on-chip routers have small buffers to minimize area overheads, thus v

and k are much smaller than in macro networks. The necessity for ultra-low latency leads to a parallel FIFO buffer design as shown in Figure 4.4. Contrary to a serial FIFO implementation, the parallel structure eliminates unnecessary intermediate processes for a flit to traverse all buffer entries until it leaves the buffer [72]. This fine-grained control requires more complex logic, which manages read and write pointers to keep the FIFO order. The read and write pointers in the parallel FIFO registers control an input demultiplexer and an output multiplexer. The write pointer points to the tail of the queue, and the read pointer points to the head of the queue. For a read operation, the flit pointed by the head is selected and transmitted to a crossbar input port. Similarly, write operation leads the incoming flit to be written to the location pointed by the tail pointer. The pointers are promptly updated after each read or write operation. After a read operation, once the head is overlapped with the tail, the buffer becomes empty. After a write operation, likewise, if the tail moves to the same position pointed by the head, the buffer is full.

4.4.2 An On-Chip Router Architecture with Hybrid Buffer Design

In this section, we show an on-chip router architecture with hybrid buffer design that combines SRAM and STT-MRAM. The hybrid design aims to maximize advantages inherent in different memory technologies in a synergistic fashion for performance improvement while consuming power economically. The key idea is inspired by the nature of STT-MRAM that provides 4 times more buffer space than SRAM under the same area constraint due to its higher density characteristics [61], [73]. The increased buffer size contributes to making on-chip routers have spacious rooms for buffering, thus boosting the overall network throughput with no additional area overheads compared to a pure SRAM-based input buffer.

Figure 4.4b depicts the proposed hybrid input buffer of a VC. Compared to the pure SRAM buffer shown in Figure 4.4a, the STT-MRAM is attached to each VC in parallel with the SRAM buffer. Each SRAM buffer entry is connected to m dedicated STT-MRAM buffer entries through separate migration links. The hybrid parallel FIFO buffer maintains read/write pointers. An incoming flit is first written to the SRAM buffer, thus the write pointer points to SRAM buffer entries only. But an outgoing flit may leave from either SRAM or STT-MRAM and the read pointer covers the entire buffer, both SRAM and STT-MRAM buffer entries.

A migration controller triggers the flit migration and determines if a certain flit is ready to be migrated to STT-MRAM. VC flow control is performed based on the availability of SRAM in downstream routers, meaning that the availability of STT-MRAM is not considered, because a write operation to STT-MRAM cannot finish in a single cycle.

Simple Flit Migration Scheme. The key design goal of the hybrid input buffer is to guarantee seamless read and write operations in every cycle to achieve higher throughput with an increased buffer size. To serve this purpose, we devise a flit migration scheme, which seamlessly migrates buffered flits from SRAM to STT-

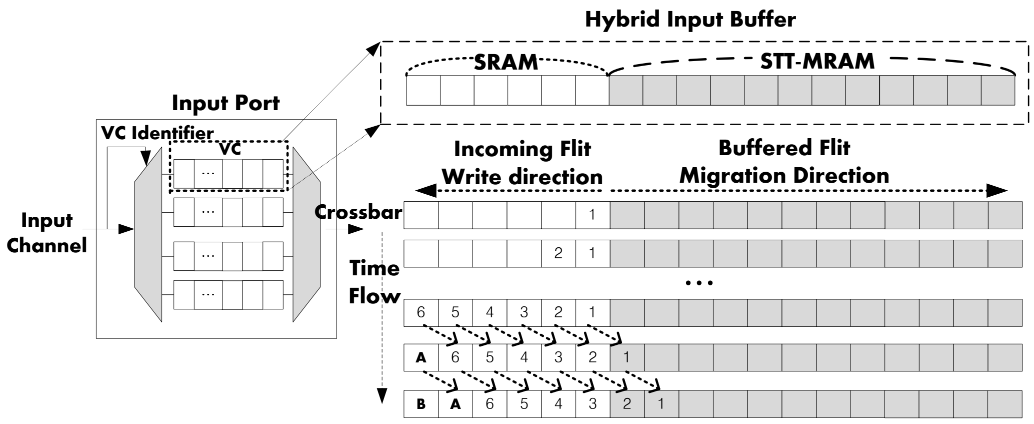


Fig. 4.5.: Simple flit migration scheme in hybrid buffer design

MRAM to secure more SRAM buffer space for incoming flits, while hiding the long write latency of STT-MRAM.

Figure 4.5 depicts an example of the migration scheme, where each VC consists of 6 SRAM and 12 STT-MRAM buffer entries. The STT-MRAM buffer write latency is assumed to be 6 cycles. When an incoming flit arrives, it is written to the SRAM buffer first, and the migration from SRAM to STT-MRAM begins immediately. Supposing that a new flit arrives every cycle, the SRAM buffer becomes full eventually in the 6th cycle. At the same time, the first flit is migrated to STT-MRAM successfully and one SRAM buffer entry becomes available. Then a subsequent incoming flit occupies the released SRAM buffer entry with no additional timing delay. Note that Figure 4.5 illustrates the concept in a logical way, and no physical shift occurs except the migration from SRAM to STT-MRAM. The placement of flits in STT-MRAM is logical and is not the physical placement described in Figure 4.4b.

Power-Efficient Lazy Migration. In the simple migration scheme explained in the previous section, the migration begins immediately as soon as an incoming flit arrives at the SRAM buffer. The simple migration wastes lots of power in a low network load because most of the flits initially written to SRAM leave the buffer in the middle of migration to STT-MRAM.

Based on this observation, we propose a **lazy migration scheme**, which selectively triggers the migration of a flit based on the estimated network load per VC in the on-chip router. The network load is indirectly estimated by tracking the number of flits in the SRAM buffer. If the ratio of the number of flits in the SRAM buffer to the total SRAM buffer size exceeds a certain predefined threshold level, the flit migration is performed for every subsequent incoming flit as long as the the ratio exceeds the threshold. In this way, we can save total write power associated with the migration operation. To implement the lazy migration scheme, the migration controller is augmented to keep track of the flits in the SRAM buffer and triggers the migration adaptively. The write power is reduced by up to 79% in a low network load compared to the simple migration, which will be discussed in detail in Secton 4.5.

4.5 Performance Evaluation

In this section, we evaluate the proposed hybrid on-chip router to examine how much it improves the overall network performance while reducing the power consumption in NoC, using several benchmarks and synthetic workloads.

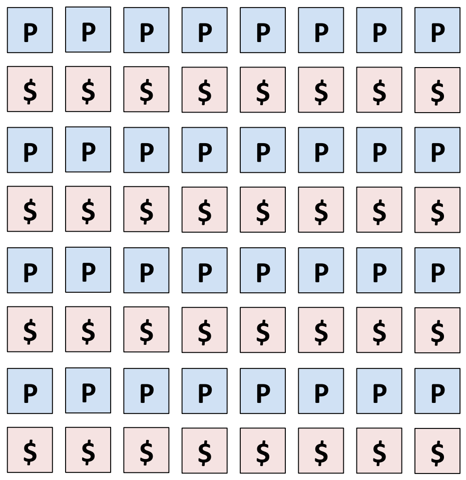


Fig. 4.6.: CMP layout

Table 4.1: CMP system configuration

System Parameters	Details
Clock frequency	3GHz
# of processors	32
L1 I and D caches	direct-mapped 32KB (L1I) 4-way 32KB (L1D), 1 cycle
L2 cache	16-way 16MB, 20 cycles 32 banks, 512 KB/bank
Cache block size	64B
Coherence protocol	Directory-based MSI
Memory latency	300 cycles
Flit size	16B
Packet size	1 flit (Benchmark-control) 5 flits (Benchmark-data) 4 flits (Synthetic)

Table 4.2: SRAM and STT-MRAM parameters

Parameter	SRAM	STT-MRAM
Read Energy (pJ/flit)	5.25	3.826
Write Energy (pJ/flit)	5.25	40.0
Leakage Power (mW)	0.028	0.005

4.5.1 System Configuration

A cycle-accurate NoC simulator is used to conduct the detailed evaluation of the proposed scheme. It implements the pipelined router architecture with VCs, a VC arbiter, a switch arbiter and a crossbar. Under the 32nm process technology, all simulations are performed in an 8x8 network having 32 out-of-order processors and 32 L2 cache banks on a single chip as shown in Figure 4.6. The network is equipped with 2-stage speculative routers with lookahead routing [69]. The router has a set of v VCs per input port. Each VC contains a k-flit buffer with 16B flit size. In our evaluation, we assume that v is 4, and k may vary with different buffer configurations. A dimension order routing algorithm, XY, and O1TURN [74] are used with wormhole switching flow control.

A variety of synthetic workloads are used to measure the effectiveness of the hybrid on-chip router: uniform random **(UR)**, bit complement **(BC)** and nearest neighbor **(NN)**. To evaluate the proposed schemes under realistic environments, we also use SPLASH-2 [28] parallel benchmark traces. The traces are obtained using Simics [25], a full system simulation platform. Table 4.1 specifies the detailed CMP configuration we use to run benchmarks.

We use Orion 2.0 [56] to estimate router power consumption. In addition, parameters shown in Table 4.2 are cited from [67], [61], for both SRAM and STT-MRAM. The unit of parameter for the leakage power is mW per 1-flit buffer. Throughout this study, the size of SRAM and STT-MRAM buffers are denoted by $SRAM\#$ and

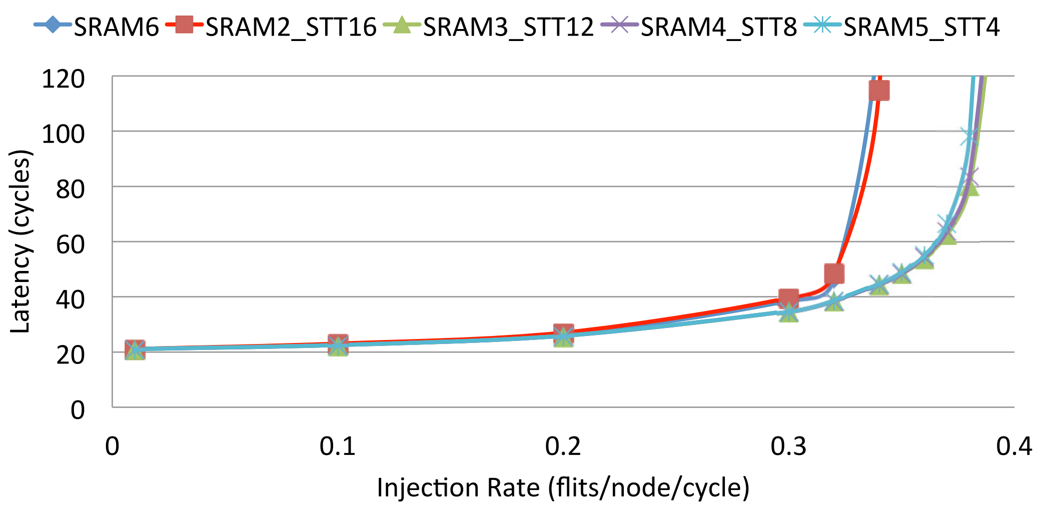
STT#, respectively. As stated in Section 4.4.2, STT-MRAM provides 4 times more buffer space compared to SRAM under the same area budget, thus *SRAM1* is equal to *STT4*. Unless otherwise stated, the write latency of STT-MRAM is 6 cycles based on the analysis in Section 4.3.

4.5.2 Performance Analysis with Synthetic Workloads and Benchmarks

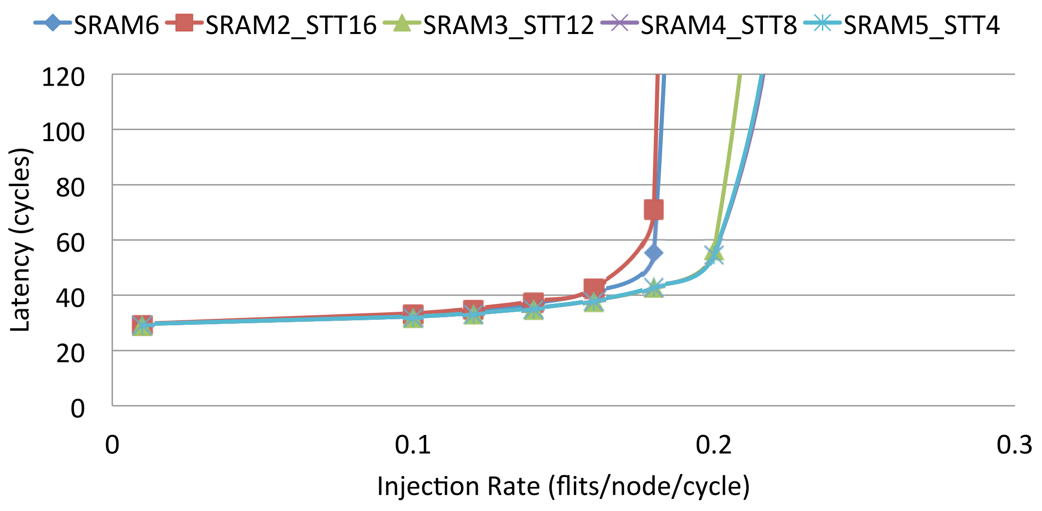
Figure 4.7 shows performance improvement for various hybrid input buffer configurations compared to the pure SRAM buffer, under UR, BC and NN traffic patterns. All results are measured under the same area budget, *SRAM6* per VC, for input buffers. In all cases, the hybrid design shows throughput improvement by 18% for UR, 28% for BC, and 17% for NN on average. These results indicate that although the STT-MRAM write latency is longer than that of SRAM, the performance loss is offset by the increased buffer size due to the high density of STT-MRAM, thus resulting in performance improvement.

We also evaluate the hybrid design using O1TURN [74] routing algorithm as well as various topologies: 2D-torus and flattened butterfly [75]. Figure 4.8 shows the performance with O1TURN in the 8x8 2D-mesh topology, where the overall throughput increases by 15% on average, while Figure 4.9 shows that the throughput is increased in 2D-torus and flattened butterfly by 13% and 15%, respectively.

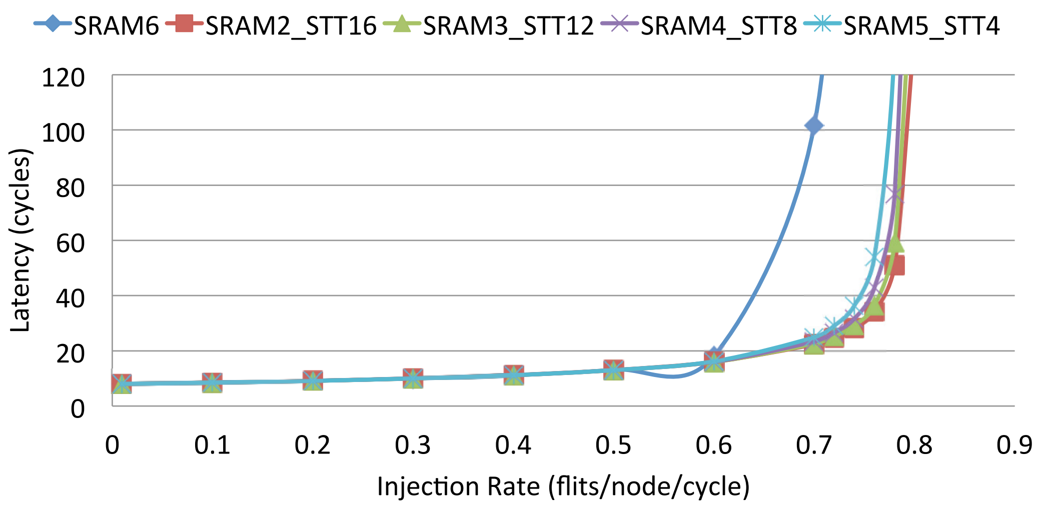
To examine the impact of different write latencies of STT-MRAM on network performance, we conduct experiments under 2D-mesh and XY routing algorithm.



(a) UR



(b) BC



(c) NN

Fig. 4.7.: Performance comparison with synthetic workloads

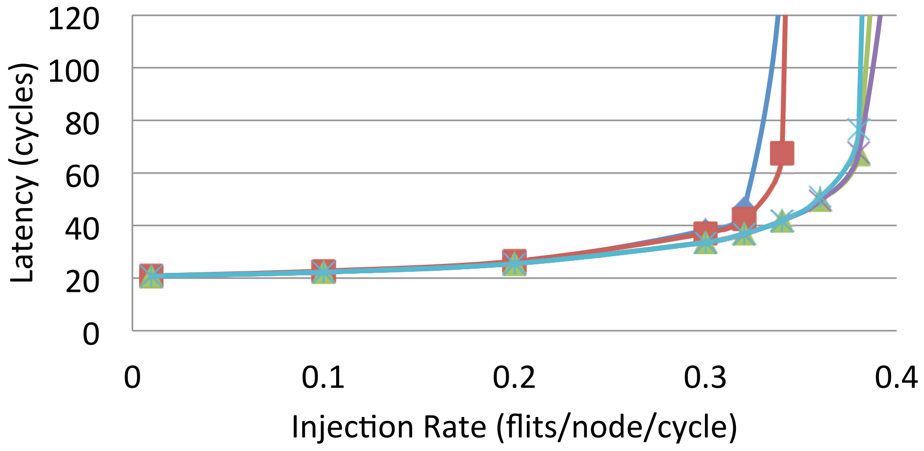
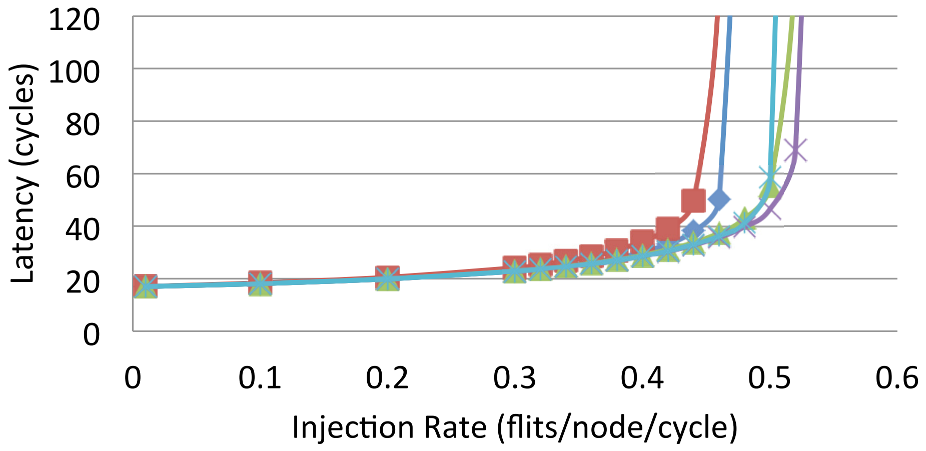
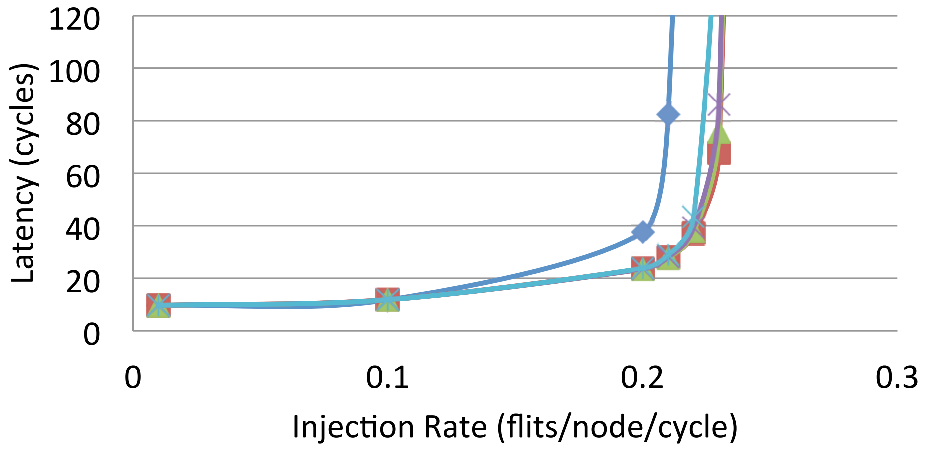


Fig. 4.8.: Performance comparison with O1TURN routing algorithm

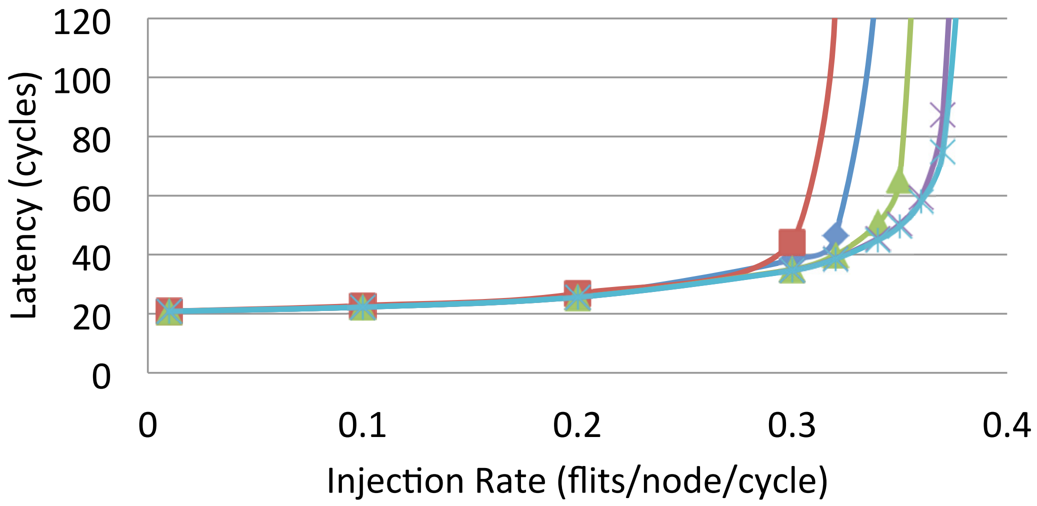


(a) 2D-Torus

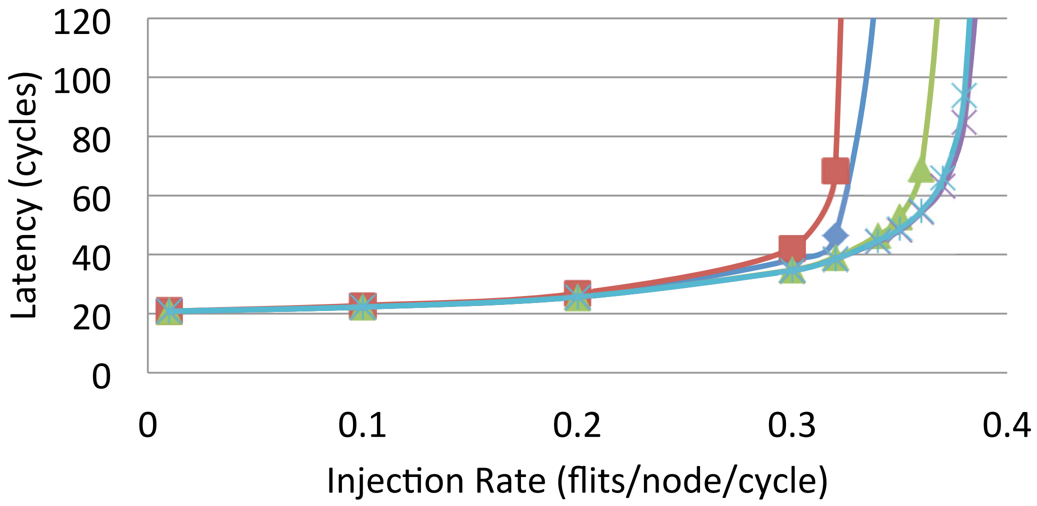


(b) Flattened Butterfly

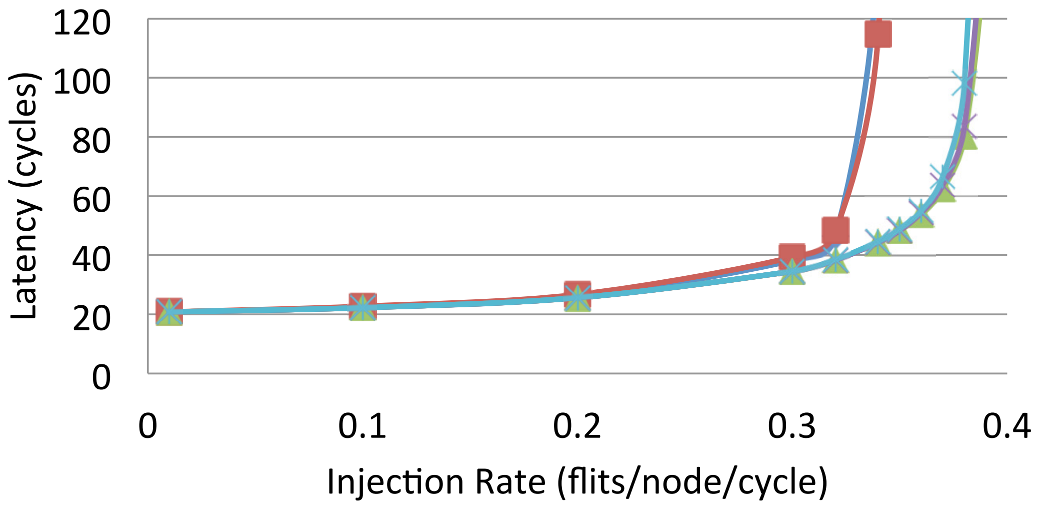
Fig. 4.9.: Performance comparison with different topologies



(a) 30 cycles



(b) 10 cycles



(c) 6 cycles

Fig. 4.10.: Performance comparsion with various STT-MRAM write latencies

Figure 4.10 shows the performance in terms of packet latency with 3 different write latencies of STT-MRAM: 30, 10, and 6 cycles. It clearly indicates that the overall network performance is affected by the duration of STT-MRAM write operation. Among the different hybrid configurations, *SRAM2_STT16* shows the worst performance. This is because the SRAM buffer space is too small to retain the incoming flits for sufficient period of time for migration, 6 cycles, which makes the simple flit migration scheme less efficient. Thus, the long write latency of STT-MRAM is not effectively hidden, resulting in the early saturation of the network. As shown in Figure 4.2, every flit stays in the buffer for at least 3 cycles. So the SRAM buffer size should be greater than or equal to 3 to run the migration scheme seamlessly.

If the write latency is long, 30 cycles, the performance is mostly determined by the SRAM size. This is because the long write latency lowers the possibility for flits to be migrated to the STT-MRAM buffer before network saturation. Therefore, *SRAM5_STT4* shows the best throughput improvement. On the contrary, if the write latency is sufficiently short, 6 cycles, the performance is greatly impacted by the total buffer size including both SRAM and STT-MRAM except the *SRAM2_STT16* case. Thus, *SRAM3_STT12* shows the highest throughput compared to other configurations.

To make a clear quantitative comparison of relative performance of the 3 different write latencies, we show network throughput normalized to the *SRAM6* in Figure 4.11, based on the results in Figure 4.10. Figure 4.11 confirms the afore-

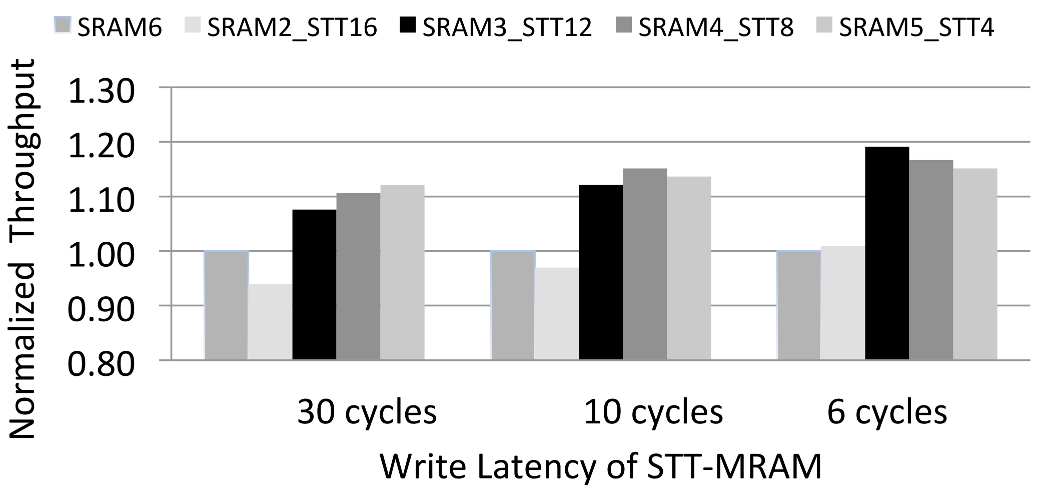


Fig. 4.11.: Throughput with different STT-MRAM write latencies

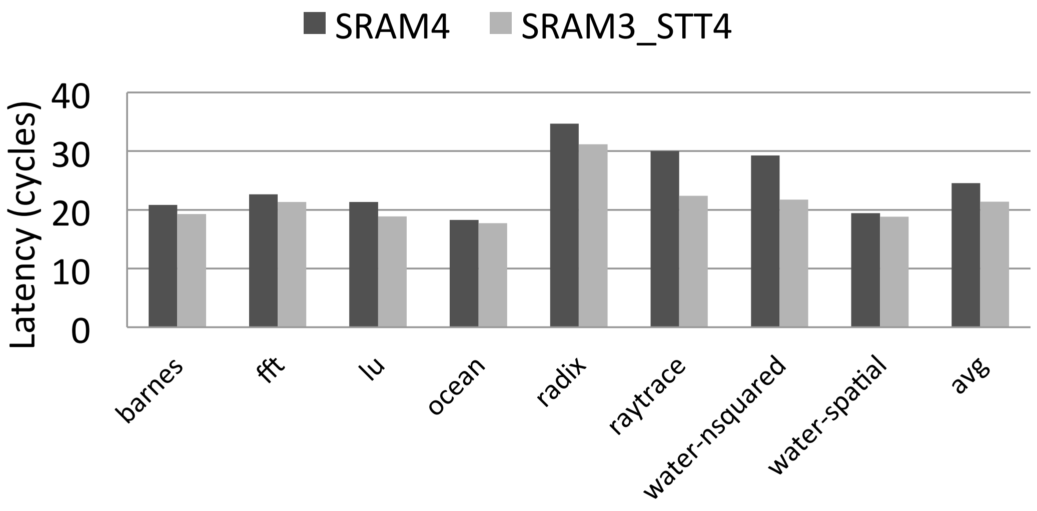
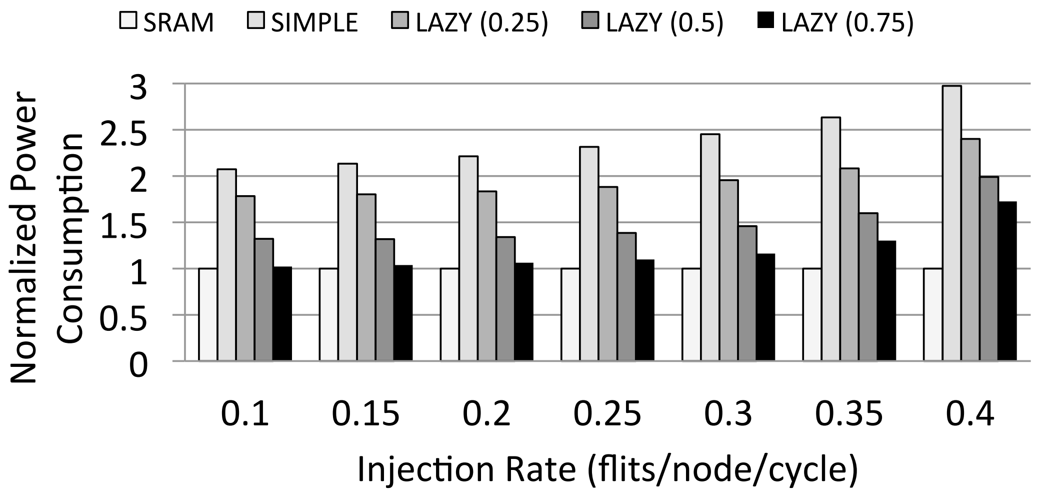


Fig. 4.12.: SPLASH-2 benchmark results

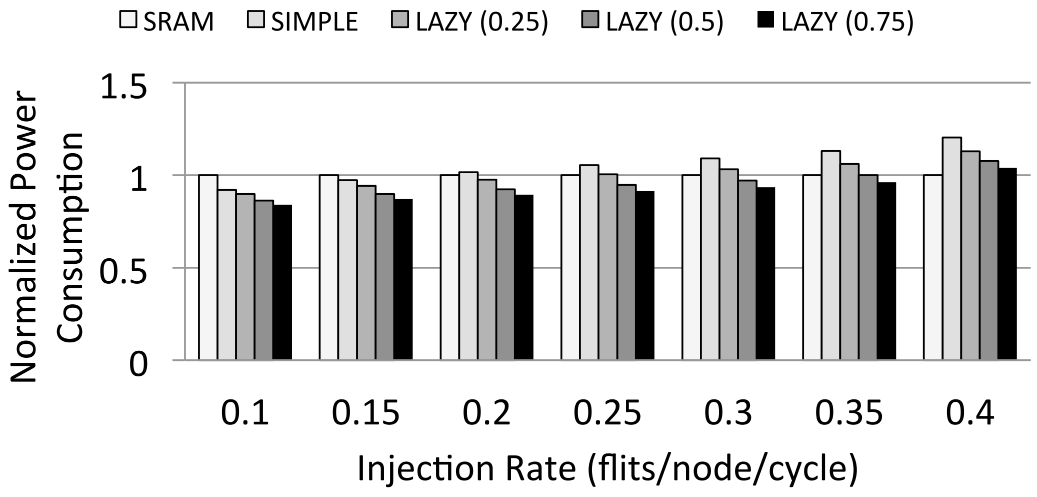
mentioned analysis. In case of a relatively long write latency, 30 cycles, the hybrid input buffer having the largest SRAM buffer outperforms the others by up to 11% compared to the pure *SRAM6* buffer. Likewise, in case of a low write latency, 6 cycles, except the *SRAM2_STT16* case, the one having the largest total buffer size, *SRAM3_STT12* beats the other configurations by up to 18% in terms of network throughput.

Figure 4.12 shows the average network latency with SPLASH-2 benchmark traces. We assume *SRAM4* per VC as an area budget, the same as a cache block size. In general, the hybrid input buffer outperforms the pure SRAM-based one, by approximately 14% on average. Specifically, *water-nsquared* shows the best improvement by 34.5% while *ocean* shows the least improvement by 3.2%. The amount of improvement varies depending on the traffic patterns. We observe that in the benchmarks showing higher improvement, hot spots exist in their communication, whereas in the benchmarks with slight performance improvement, communication is evenly spread across the whole network.

Finally, we make a sensitivity analysis of the number of buffer entries in NoC routers. Under two different area budgets, *SRAM4* and *SRAM6*, we compare the throughput of the pure SRAM-based buffer and the hybrid buffer that shows the best performance. As the budget decreases from *SRAM6* to *SRAM4*, the amount of improvement coming from the hybrid buffer increases by approximately 5.5%. This trend indicates that the hybrid buffer is more beneficial as the area budget in CMP environments becomes tighter.



(a) Dynamic power consumption of input buffers



(b) Total power consumption of routers

Fig. 4.13.: Comparison of power efficiency

4.5.3 Power Analysis

Since power is one of the main issues in the NoC router design, we evaluate power consumption of the hybrid input buffer and compare the effect of the two migration schemes explained in Section 4.4. Figure 4.13a compares the dynamic buffer power consumption of 4 different migration schemes in *SRAM3_STT12*: simple and lazy

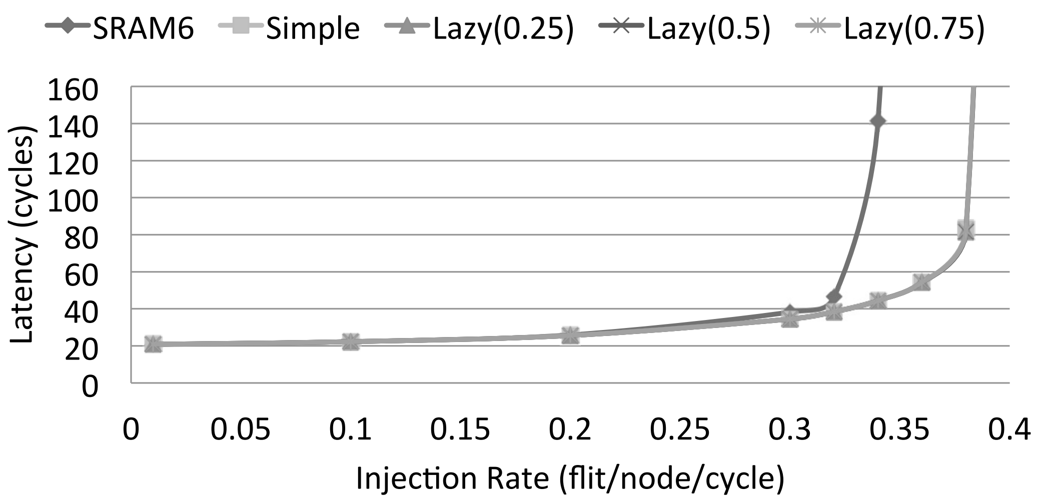


Fig. 4.14.: Performance comparison with different threshold in lazy migration

with 3 different thresholds (0.25/0.5/0.75). All results are normalized to that of the pure SRAM-based buffer, *SRAM6*. The lazy migration scheme with the threshold 0.75 consumes significantly less amount of power, by 53% on average, compared to the simple migration scheme. In a low network load (0.1), the power consumption of the lazy migration scheme with the threshold 0.75 is almost equivalent to that of the baseline SRAM. In a high network load (0.4), however, the flit migration occurs more frequently in the hybrid buffer due to the highly congested network. Accordingly, the migration lowers the possibility of reducing the dynamic power, thus increasing the power consumption of the lazy migration by up to 1.7x more than the baseline SRAM.

Figure 4.13b compares the total router power consumption of the 4 migration schemes that includes both leakage and dynamic power consumption of all routers across the network. In a low network load (0.1), the total power consumption of routers with the hybrid buffer is less than that of routers with the pure SRAM buffer by 16%. This is due to much less leakage power consumption of STT-MRAM compared to SRAM as shown in Table 4.2. As the network gets more congested, however, the hybrid buffer consumes more power compared to the baseline SRAM buffer. In a high network load (0.4), for instance, the lazy migration scheme with the threshold 0.75 consumes more power by up to 4% compared to the baseline SRAM buffer.

In order to show the effect on the performance in detail, we compare the performance of simple and lazy migration schemes with pure SRAM under the same area budget, *SRAM6*, in Figure 4.14. Both the simple and lazy schemes outperform the pure SRAM. Also, as we increase the threshold value from 0.25 to 0.75 in the lazy migration scheme, the overall network throughput is slightly degraded but the amount of degradation is around 0.5% on average, which is negligible.

4.6 Conclusions

In this study, we have proposed a hybrid input buffer design using STT-MRAM with SRAM to achieve better network throughput with marginal power overheads in on-chip interconnection networks. The high density of STT-MRAM facilitates to accommodate larger buffer compared to the conventional SRAM under the same area budgets. Through the flit migration schemes, the long write latency of STT-MRAM is effectively hidden while minimizing the power overheads. Simulation results indicate performance improvement of around 21% and 14% on average under the synthetic workloads and benchmarks, respectively, compared to the conventional on-chip router with the SRAM input buffer.

For future work, we intend to devise an STT-MRAM-aware routing algorithm and provide an architectural support to reduce the overall power consumption and latency further.

5. CONCLUSIONS

Even though the HPC systems using CMP-based multiprocessors are beneficial to meet the high performance requirements, there are a number of issues to be resolved, such as networking performance of both off-chip and on-chip interconnects, power efficiency and security. To address these issues, three schemes have been proposed to design high-performance, power-efficient and secure HPC systems.

First, we have proposed VSC to reduce the overheads of off-chip interconnects by decreasing the number of packets as well as the packet size. The packet compression technique, VSC, works in harmony with an underlying cache coherence mechanism and achieves significant performance improvement by cancelling packet transmission for the most frequent data known to all nodes. Eliminating data packets tremendously reduces cache miss latency, which enhances overall system performance. The hybrid counter management achieves perfect counter prediction with low storage overhead by using global and per-block counters together, which allows data packets not to carry counters for encryption.

Second, we have proposed a fast and efficient bounds checking mechanism for multi-threaded workloads in CMP systems. The bounds information sharing reduces the space overheads for storing bounds information and the smart tagging enables the skipping of bounds checking for pointers already guaranteed to be safe. The BCache architecture allows fast delivery of bounds information as well as regular data to a requestor node by duplicating the same data block that might be shared by threads in multiple locations.

Third, we have proposed a hybrid input buffer design using STT-MRAM with SRAM to achieve better network throughput with marginal power overheads in on-chip interconnection networks. The high density of STT-MRAM facilitates to accommodate larger buffer compared to the conventional SRAM under the same area budgets. Through the flit migration schemes, the long write latency of STT-MRAM is effectively hidden while minimizing the power overheads.

REFERENCES

[1] "Infiniband Architecture Specification, Volume 1, Release 1.1," *InfiniBand Trade Association*, 2002.

[2] R.Das, A.K.Mishra, C.Nicopoulos, D.Park, V.Narayanan, R.Iyer, M.S.Yousif, and C.R.Das, "Performance and Power Optimization through Data Compression in Network-on-Chip Architectures," *Proceedings of HPCA*, 2008.

[3] Y.Jin, K.H.Yum, and E.J.Kim, "Adaptive Data Compression for High-Performance Low-Power On-Chip Networks," *Proceedings of MICRO*, 2008.

[4] C.Yan, B.Rogers, D.Englender, Y.Solihin, and M.Prvulovic, "Improving Cost, Performance, and Security of Memory Encryption and Authentication," *Proceedings of ISCA*, 2006.

[5] M.Lee, M.Ahn, and E.J.Kim, "I^{2}SEMS: Interconnects-Independent Security Enhanced Shared Memory Multiprocessor Systems," *Proceedings of PACT*, 2007.

[6] B.Rogers, M.Prvulovic, and Y.Solihin, "Efficient Data Protection for Distributed Shared Memory Multiprocessors," *Proceedings of PACT*, 2006.

[7] B.Rogers, S.Chhabra, Y.Solihin, and M.Prvulovic, "Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly," *Proceedings of MICRO*, 2007.

[8] B.Rogers, C.Yan, S.Chhabra, M.Prvulovic, and Y.Solihin, "Single-Level Integrity and Confidentiality Protection for Distributed Shared Memory Multiprocessors," *Proceedings of HPCA*, 2008.

[9] J.Yang, Y.Zhang, and L.Gao, "Fast Secure Processor for Inhibiting Software Piracy and Tampering," *Proceedings of MICRO*, 2003.

[10] Y.Zhang, L.Gao, J.Yang, X.Zhang, and R.Gupta, "SENSS: Security Enhancement to Symmetric Shared Memory Multiprocessors," *Proceedings of HPCA*, 2005.

[11] A.R.Alameldeen and D.A.Wood, "Adaptive Cache Compression for High-Performance Processors," *Proceedings of ISCA*, 2004.

[12] D. Citron and L. Rudolph, "Creating A Wider Bus Using Caching Techniques," *Proceedings of HPCA*, 1995.

[13] M.Farrens and A.Park, "Dynamic Base Register Caching: A Technique for Reducing Address Bus Width," *Proceedings of ISCA*, 1991.

[14] J.-S.Lee, W.-K.Hong, and S.-D.Kim, "Design and Evaluation of A Selective Compressed Memory System," *Proceedings of ICCD*, 1999.

[15] M.H.Lipasti, C.B.Wilkerson, and J.P.Shen, "Value Locality and Load Value Prediction," *Proceedings of ASPLOS*, 2007.

[16] R.B.Tremaine, T.B.Smith, M.Wazlowski, D.Har, K.-K.Mak, and S.Arramreddy, "Pinnacle: IBM MXT in A Memory Controller Chip," *IEEE Micro*, vol. 21, pp. 56–68, 2001.

[17] Y.Zhang, J.Yang, and R.Gupta, "Frequent Value Locality and Value-Centric Data Cache Design," *Proceedings of ASPLOS*, 2000.

[18] D.Lie, C.Thekkath, M.Mitchell, P.Lincoln, D.Boneh, J.Mitchell, and M.Horowitz, "Architectural Support for Copy and Tamper Resistant Software," *Proceedings of ASPLOS*, 2000.

[19] G.E.Suh, D.Clarke, B.Gassend, M. Dijk, and S.Devadas, "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," *Proceedings of ICS*, 2003.

[20] W.Shi, H.S.Lee, M.Ghosh, C.Lu, and A.Boldyreva, "High Efficiency Counter Mode Security Architecture via Prediction and Precomputation," *Proceedings of ISCA*, 2005.

[21] G.E.Suh, D.Clarke, B.Gassend, M. Dijk, and S.Devadas, "Efficient Memory Integrity Verification and Encryption for Secure Processors," *Proceedings of MICRO*, 2003.

[22] G.E.Suh, C.W.O'Donnell, I.Sachdev, and S.Devadas, "Design and Implementation of The AEGIS Single-Chip Secure Processor Using Physical Random Functions," *Proceedings of ISCA*, 2005.

[23] B.Gassend, G.E.Suh, D.Clarke, M. Dijk, and S.Devadas, "Caches and Hash Trees for Efficient Memory Integrity Verification," *Proceedings of HPCA*, 2003.

[24] W.Shi, H.-H.S.Lee, M.Ghosh, and C.Lu, "Architectural Support for High Speed Protection of Memory Integrity and Confidentiality in Multiprocessor Systems," *Proceedings of PACT*, 2004.

[25] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, and B. Werner, "Simics: A Full System Simulation Platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[26] M.M.Martin, D.J.Sorin, B.M.Beckmann, M.R.Marty, M.Xu, A.R.Alameldeen, K.E.Moore, M.D.Hill, and D.A.Wood, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *Computer Architecture News(CAN)*, 2005.

[27] T.Kgil, L.Falk, and T.Mudge, "Chiplock: Support for Secure Microarchitectures," *Proceedings of the Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, Oct 2004.

[28] S.C.Woo, M.Ohara, E.Torrie, J.P.Singh, and A.Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proceedings of ISCA*, 1995.

[29] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," *Proceedings of PACT*, 2008.

[30] R. Hastings and B. Joyce, "Purify: Fast Detection of Memory Leaks and Access Errors," *Proceedings of the Winter 1992 USENIX Conference*, 1991.

[31] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," *Proceedings of PLDI*, 2007.

[32] Z. Shao, J. Cao, K. C. C. Chan, C. Xue, and E. H.-M. Sha, "Hardware/Software Optimization for Array & Pointer Boundary Checking Against Buffer Overflow Attacks," *Journal of Parallel and Distributed Computing - Special issue: Security in grid and distributed systems*, vol. 66, pp. 1129–1136, September 2006.

[33] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic, "MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging," *Proceedings of HPCA*, 2007.

[34] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: Architectural Support for Spatial Safety of the C Programming Language," *Proceedings of ASPLOS*, 2008.

[35] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood, "Crafting a Usable Microkernel, Processor, and I/O System with Strict and Provable Information Flow Security," *Proceeding of ISCA*, 2011.

[36] S. Chhabra and Y. Solihin, "i-NVMM: A Secure Non-Volatile Main Memory System with Incremental Encryption," *Proceeding of ISCA*, 2011.

[37] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient Detection of All Pointer and Array Access Errors," *Proceedings of PLDI*, 1994.

[38] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly Compatible and Complete Spatial Memory Safety for C," *Proceedings of PLDI*, 2009.

[39] G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-Safe Retrofitting of Legacy Code," *Proceedings of POPL*, 2002.

[40] N. Nethercote, "Bounds-Checking Entire Programs Without Recompiling," *Proceedings of SPACE*, 2004.

[41] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic, "Comprehensively and Efficiently Protecting the Heap," *Proceedings of ASPLOS*, 2006.

[42] E. D. Berger and B. G. Zorn, "Diehard: Probabilistic Memory Safety for Unsafe Languages," *Proceedings of PLDI*, 2006.

[43] W. Chuang, S. Narayanasamy, B. Calder, and R. Jhala, "Bounds Checking with Taint-Based Analysis," *Proceedings of HiPEAC*, 2007.

[44] T.-c. Chiueh, "Fast Bounds Checking Using Debug Register," *Proceedings of HiPEAC*, 2008.

[45] D. Dhurjati and V. Adve, “Backwards-Compatible Array Bounds Checking for C with Very Low Overhead,” *Proceedings of ICSE*, 2006.

[46] P. Akritidis, M. Costa, M. Castro, and S. Hand, “Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors,” *Proceedings of USENIX Security Symposium*, 2009.

[47] R. Bodík, R. Gupta, and V. Sarkar, “ABCD: Eliminating Array Bounds Checks on Demand,” *Proceedings of PLDI*, 2000.

[48] M. Joyner, Z. Budimlić, and V. Sarkar, “Subregion Analysis and Bounds Check Elimination for High Level Arrays,” *Proceedings of CC*, 2011.

[49] W. Chuang, S. Narayanasamy, and B. Calder, “Accelerating Meta Data Checks for Software Correctness and Security,” *Journal of Instruction-Level Parallelism*, vol. 9, 2007.

[50] “System V Application Binary Interface, Edition 4.1, Chap. 4,” http://www.sco.com/developers/devspecs/gabi41.pdf.

[51] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, “CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs,” *Proceedings of the CC*, 2002.

[52] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, “CACTI 5.1,” HP Laboratories, Tech. Rep. HPL-2008-20, 2008.

[53] J. Balfour and W. J. Dally, “Design Tradeoffs for Tiled CMP On-Chip Networks,” *Proceedings of ICS*, 2006.

[54] D. Vantrease, R. Schreiber, M. Monchiero, M. McLaren, N. P. Jouppi, M. Fiorentino, A. Davis, N. Binkert, R. G. Beausoleil, and J. H. Ahn, “Corona: System Implications of Emerging Nanophotonic Technology,” *Proceedings of ISCA*, 2008.

[55] M. T. Yourst, “PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator,” *Proceedings of ISPASS*, 2007.

[56] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi, “ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-Stage Design Space Exploration,” *Proceedings of DATE*, 2009.

[57] A. Jog, A. K. Mishra, C. Xu, Y. Xie, N. Vijaykrishnan, R. Iyer, and C. R. Das, “Cache Revive: Architecting Volatile STT-RAM Caches for Enhanced Performance in CMPs,” The Pennsylvania State University CSE Dept., Tech. Rep. CSE-11-010, June 2011.

[58] C. W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan, “Relaxing Non-Volatility for Fast and Energy-Efficient STT-RAM Caches,” *Proceedings of HPCA*, 2011.

[59] A. K. Mishra, X. Dong, G. Sun, Y. Xie, N. Vijaykrishnan, and C. R. Das, “Architecting On-Chip Interconnects for Stacked 3D STT-RAM Caches in CMPs,” *Proceedings of ISCA*, 2011.

[60] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen, "A Novel Architecture of the 3D Stacked MRAM L2 Cache for CMPs," *Proceedings of HPCA*, 2009.

[61] X. Guo, E. Ipek, and T. Soyata, "Resistive Computation: Avoiding the Power Wall with Low-Leakage, STT-MRAM Based Computing," *Proceedings of ISCA*, 2010.

[62] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology," *Proceedings of ISCA*, 2009.

[63] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High Performance Main Memory System Using Phase-Change Memory Technology," *Proceedings of ISCA*, 2009.

[64] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," *Proceedings of ISCA*, 2009.

[65] M. K. Qureshi, M. M. Franceschini, and L. A. Lastras-montaño, "Improving Read Performance of Phase Change Memories via Write Cancellation and Write Pausing," *Proceedings of HPCA*, 2010.

[66] P. Zhou, Y. Du, Y. Zhang, and J. Yang, "Fine-Grained QoS Scheduling for PCM-based Main Memory Systems," *Proceedings of IPDPS*, 2010.

[67] ITRS, "International Technology Roadmap for Semiconductors: 2009 Executive Summary," http://www.itrs.net/Links/2009ITRS/Home2009.htm.

[68] L.-S. Peh and W. J. Dally, "A Delay Model and Speculative Architecture for Pipelined Routers," *Proceedings of HPCA*, 2001.

[69] M. Galles, "Scalable Pipelined Interconnect for Distributed Endpoint Routing: The SGI SPIDER Chip," *Proceedings of Hot Interconnect 4*, 2009.

[70] W. J. Dally and C. L. Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Trans. Comput.*, vol. 36, pp. 547–553, May 1987.

[71] W. J. Dally, "Virtual-Channel Flow Control," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, pp. 194–205, March 1992.

[72] A. V. Yakovlev, A. M. Koelmans, and L. Lavagno, "High-Level Modeling and Design of Asynchronous Interface Logic," *IEEE Design and Test of Computers*, vol. 12, pp. 32–40, 1995.

[73] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "Energy Reduction for STT-RAM Using Early Write Termination," *Proceedings of ICCAD*, 2009.

[74] D. Seo, A. Ali, W.-T. Lim, N. Rafique, and M. Thottethodi, "Near-Optimal Worst-Case Throughput Routing for Two-Dimensional Mesh Networks," *Proceedings of ISCA*, 2005.

[75] J. Kim, J. Balfour, and W. Dally, "Flattened Butterfly Topology for On-Chip Networks," *Proceedings of MICRO*, 2007.

VITA

Baik Song An received his Bachelor of Science degree in Computer Science and Master of Science degree in Electrical Engineering and Computer Science from Seoul National University in 1999 and 2001, respectively. Before he was admitted to Texas A&M University in August 2006, he had work experiences as a research staff at Electronics and Telecommunications Research Institute (ETRI) in South Korea. His research interests consist of computer architecture and system software including operating systems. Mr. An may be reached at 3112 TAMU, College Station, TX 77843-3112. His email is baiksong@cse.tamu.edu.