PERFORMANCE EVALUATION OF SHARED-MEMORY PROTOCOLS FACING GRAPH DATA

BY

ALIREZA NAZARI, B.S.

A thesis submitted to the Graduate School

in partial fulfillment of the requirements

for the degree

MASTER OF SCIENCE

Major Subject: ELECTRICAL ENGINEERING

NEW MEXICO STATE UNIVERSITY

LAS CRUCES, NEW MEXICO

JUNE 2014

"Performance Evaluation of Shared-Memory Protocols Facing Graph Data," a thesis prepared by Alireza Nazari in partial fulfillment of the requirements for the degree Master of Science, has been approved and accepted by the following:

Linda Lacey

Dean of the Graduate School

Jeanine Cook

Chair of the Examining Committee

Date

Committee in charge:

Dr. Jeanine Cook, Chair

Dr. David Voelz

Dr. Jonathan Cook

PERFORMANCE EVALUATION OF SHARED-MEMORY PROTOCOLS FACING GRAPH DATA

BY

ALIREZA NAZARI, B.S.

MASTER OF SCIENCE
NEW MEXICO STATE UNIVERSITY, 2014
Dr. Jeanine Cook, Chair

Memory performance has not improved at the same pace as processor performance. This fact has led to a tendency toward integrating caches in processors, making use of multiple small, high speed memories. To increase the performance of these caches, algorithms and hardware are organized to take advantage of spatial and temporal locality.

In scenarios with multiple processing units, there is additional complexity since some data has to be shared among the various processing units. A protocol is required to keep this shared data coherent. MESI protocol and its variants are the most common cache coherency protocol in multicore systems.

Graph applications are known for having low locality, and this lack of locality could be a major source of performance loss in these applications as the number of threads increases. To quantify performance loss due to memory transfer, two graph benchmarks, Graph500 and SSCA2, and one regular parallel benchmark, HeatPlate, were chosen. First, the performance loss was characterized by an instrumentation tool, PGOMP. Then, the MARSSx86 simulator was used to directly measure the

MESI overhead. Finally, results analysis showed that MESI overhead was the source of up to 80 percent of performance loss in these graph benchmarks.

DEDICATION

To my parents, brother and sister

for their endless love, support and encouragement

ACKNOWLEDGMENTS

I extend my gratitude to my academic advisor Dr. Jeanine Cook for her inspiration, continuous support, and timely guidance throughout my studies at New Mexico State University. I am fortunate to be a part of her research team, which helped me to grow not only in academics but as a complete individual. I owe my deepest gratitude for her generous financial support for my studies. I would also like to thank Dr. Waleed Alkohlani for his willingness to invest a lot of time answering questions and discussing research. Without his guidance and wisdom, this research would not have been possible.

I am thankful to my exam committee, Dr. Voelz and Dr. Jonathan Cook, for their contributions in editing this thesis, for their words of encouragement, and for their suggestions for improvement.

I am indebted to all my colleagues and friends in the ACAPS Lab for providing a stimulating and fun environment for learning and growth. I would like to convey my deep gratitude to my dearest family and my friends for all of their help, love, and support.

VITAE

1989	Born in Isfahan, Iran
2010	Undergraduate Intern IT Department, Bank of Mining & Industry, Tehran, Iran
2007-2011	B.S., Computer Engineering-Hardware Shahed University, Tehran, Iran
2012-2014	Research Assistant Advanced Computer Architecture, Performance and Simulation Laboratory (ACAPS), NMSU, USA
2012-2014	M.S., Electrical Engineering New Mexico State University, NM, USA

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

Processor performance has been improving at roughly 60% per year. Memory access time, however, has improved by less than 10% per year [1]. This uneven development in processor and memory speed has introduced an issue known as the processor-memory performance gap or *memory wall* [2]. To alleviate performance degradation due to this issue, out-of-order processors and multilevel caches have been introduced [1]. Out-of-order and speculative instruction execution attempt to hide memory latency by keeping the pipeline full with instructions that are independent of stalled memory instructions. The overhead of these techniques is more complexity and power consumption in the processing unit. Conversely, multilevel cache hierarchy provides a fast but small on-chip memory, which means a smaller miss penalty than memory access time, while hit cost is low. Consequently, we tend to keep the data that we are going to use in the near future in the highest possible cache level, near the processor. Taking limited cache capacity into account, locality plays an important role in faster computation.

In a multiprocessing system scenario, several processing units are integrated on a single chip or connected by a network. Each processor has its own private cache hierarchy and a shared cache at the lowest level of hierarchy. In this scenario, keeping data coherent between all caches and main memory is a significant issue. A coherency protocol should enable processors to communicate with each other and inform each other

about the most recent status of data. MESI and its variants are the most common coherency protocols which support write-back cache.

The cost of this guaranteed coherency of the data includes overhead on the interconnect and memory hierarchy, as well as an additional logic that implements the coherency protocol. This overhead affects memory operation performance (average access time), memory interconnect traffic, and also energy consumption. Most importantly, this overhead limits the perfect scalability and speedup of the system. Moreover, since the whole idea behind using a cache hierarchy is to take advantage of spatial/temporal locality, a low level of locality in a particular workload exacerbates the coherency overhead. In Section 3.1, temporal and spatial locality is discussed in more detail. Also, more detailed information on bottlenecks, which limit scaling and speedup, is provided in section 2.3.

1.1 MOTIVATION

In computer science and mathematics, graphs are abstract data structures that model structural relationships among objects. They are now widely used for data modeling in application domains for which identifying relationship patterns, rules, and anomalies are useful. These domains include the web graph, social networks, the Semantic Web, knowledge bases, protein-protein interaction networks, and bibliographical networks, among many others. The ever-increasing size of graph-

structured data for these applications creates a critical need for scalable systems which can efficiently process extremely large amounts of data.

Considering the large execution time of graph applications and the demand for time and energy performance, the identification of performance bottlenecks is important. One of the inherent attributes of graph applications is the lack of the data locality. Consequently, a large amount of inter-processor data transfer is required to keep the data coherent. In this work, we quantify overhead associated with coherency in graph-based applications using a simulation. We attempt to show that a significant portion of total execution time is spent on coherence and coherence-related events, particularly for graph applications that are characterized by little spatial and temporal locality. In Section 2.1 we focus on the MESI protocol in more detail and see why it forces overhead to parallel applications.

Chapter 3 provides a comprehensive literature review on the application performance characterization of parallel application and overhead characterization. Then, in Section 3.3, we focus on literatures which attempt to characterize the performance of graph applications, and find that the literature does not provide enough profiling data and insight about performance characterization. An especial gap is the role of coherency protocol in poor speedup of graph applications. Therefore, we tried to provide more insight about performance bottlenecks and their effects on scaling by quantifying the influence of coherency protocol influence on speedup. In Chapter 4 , we define scaling

overheads in general and coherency overhead. Then, we explain why we have to use an accurate cycle simulator to measure coherency overhead. The methodology and more specific details about the simulator are discussed. The real-world machine and the simulator are fully explained, and the two are compared to show their commonalities and differences. In Section 4.4, our coherency overhead measurement in the simulator is fully explained. Also, the examined graph benchmarks are briefly described in section 4.5.

Finally, the achieved results are presented in Chapter 5 . This chapter starts with the locality evaluation of benchmarks. We use the metrics described in Section 2.1 to show that the examined graph benchmarks have a low level of locality. In Section 5.3, an instrumentation tool, PGOMP, is used to profile barrier, critical section, lock, and coherency protocol overhead. Then, in section 5.4, coherency protocol overhead is directly measured, characterized, and compared to PGOMP results. We show how many cycles are wasted by the MESI protocol and what fraction of wasted speedup is due to this overhead. In Chapter 6 , the conclusion is made and more applications of this analysis are discussed.

2 BACKGROUND

2.1 MEMORY COHERENCY

The existing techniques of increasing *instruction level parallelism* (ILP) are no longer able to track the performance and speed that Moor's Law suggests. Energy, heat, and wire delay issues are the obstacles that obstruct the expected performance track [3]. Therefore, processor vendors are now focusing on *thread-level parallelism* (TLP) by designing chips with multiple processors, known as *Multicore* or *Chip-level Multiprocessors* (CMP). By extracting higher-level TLP on multicores, performance can continue to improve. However, managing the technology issues, which are faced by increasing the performance of conventional single-core designs, is a problem [4]. A brief look at multiprocessor development trends shows an exponential increase in on-chip cores. This fast growth, combined with the growth rate of *Moore's Law*, suggests the possibility that thousand-core CMPs may be produced in the near future [5].

The shift toward multicore processors depends on parallel software and the *shared-memory model* to achieve continued exponential performance gains. In the *shared-memory model*, all processors access the same physical address space. Since each processor has its own private cache hierarchy, copies of the same data are present in different caches at the same time. Therefore, a major problem in multiprocessors is providing a consistent view of memory for each processor. The cache coherence problem is a critical, performance-sensitive design point for supporting the shared-memory model.

Cache coherence mechanism should take care of (*i*) communication between processors and (*ii*) how the data transfers between the processors, caches and memory. Assuming the shared memory programming model remains prominent, future workloads will depend upon the performance of the cache coherent memory system [4]. Cache coherence protocol is a distributed algorithm, which is used to maintain coherency among all of the data copies. Various cache coherency protocols have been introduced [6, 7, 8]. The major difference between these protocols is in the performed action on a write.

2.2 PROTOCOL STRATEGIES

Cache coherency protocols can implement two different strategies depending on how each processor informs the other processors about modifications in its local cache. It can either *invalidate* the stale data and wait for next read to update, or send an *update* instantly. Furthermore, the protocol should implement a writing policy. In *write through*, memory is updated whenever a processor performs a write. In *write-back*, the memory can be updated in two ways: first, when another processor reads the same cache block; and second, when a processor with the only valid copy of the block replaces it. The second condition happens when the cache needs to evict the cache block. Making a correct decision about strategy can affect performance dramatically. The write-back invalidate approach is the mainstream approach in cache coherency protocols, but since the best approach depends on application, it is not always the best solution.

2.2.1 PROTOCOL IMPLEMENTATION

In every read/write access, a permission should be checked to see whether a cache block is accessible for that memory operation or not. At any point in logical time, the permissions for a cache block can allow either a single writer or multiple readers. This permission mechanism is implemented by a set of cooperating finite state machines. So, for each defined access granularity, the hardware-implemented finite state machine checks certain conditions and performs the required action to keep that granularity coherent. The appropriate action is selected based on (*i*) issued memory operation and (*ii*) the state of the machine. The cache coherence is implemented in two schemes, *snoopy* and *directory-based*. To explain the reason behind the existence of these two schemes, understanding about different classes of multiprocessors is required.

Symmetric Multiprocessors (SMP): In a SMP machine, the access latency of all memory space is the same (Figure 1). A multicore is an SMP system in which every core has access to the IO and memory and is treated equally by one common OS instance. Communication between caches and memory is achieved by using a broadcast mechanism.

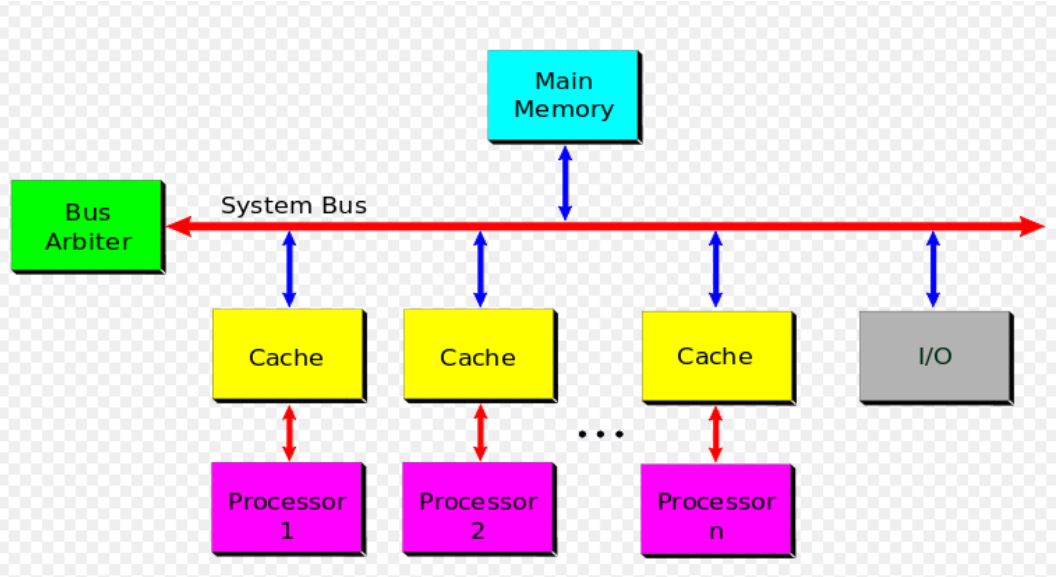

Figure 1- SMP Machine

Non-Uniform Memory Access (NUMA): In a NUMA machine, the memory access time depends on the memory location relative to the processor (Figure 3). As an extension to this definition, *ccNUMA* is a NUMA approach that takes advantage of coherent caches. An implemented *Distributed Shared Memory (DSM)* machine provides a single logical address space for all processors, as well as guaranteed coherent cache.

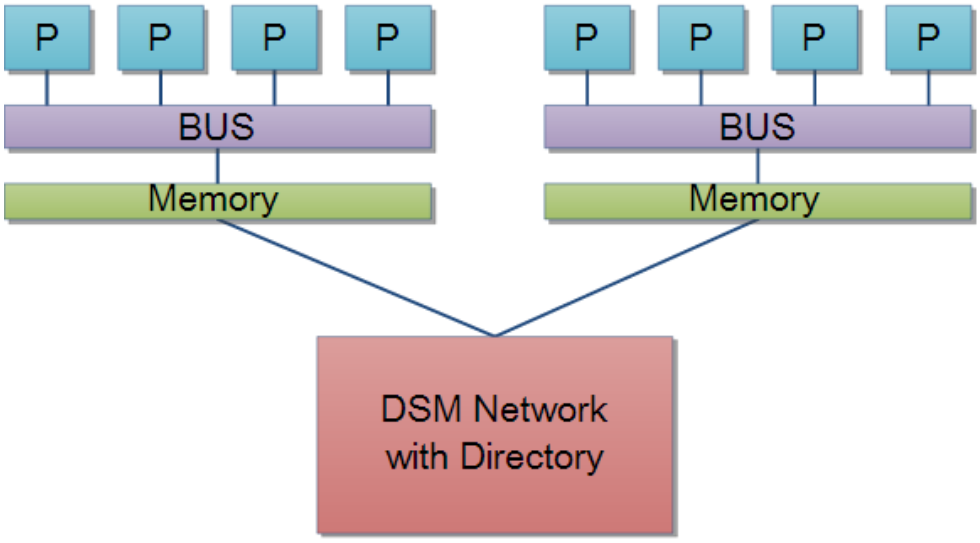

Figure 2- ccNUMA Machine

In an SMP processor, assumption is visible traffic for each core. This means coherency messages are sent by broadcasting on the bus. By utilizing this feature, handling the protocol requires fewer hardware resources and can be implemented with relatively lower cost and space. Each core snoops the interconnect network, and changes the current state of state-machines based on the received messages. Obviously cores ignore a message if the corresponding memory block is not in their cache. This method is called *snoopy cache*.

In DSM systems, processors (each processor can be multicore SMP) connect to each other by an inter-processors network. This network is a scalable network that uses multiple components. So, by broadcasting coherency messages on the network, the performance plunges. To have a direct access to take each memory block location instead

of broadcasting, an additional logic unit is required. This unit is called *directory*. In larger scales of on-chip processors, a combination of snoopy and directory-based approaches is used [9, 10]. This combination takes advantage of the lower request latency associated with snoopy protocol and the bandwidths savings associated with directory-based protocols. The decision between these two protocols is made in real-time based on recent network statistics.

2.2.2 IMPLEMENTATION OF SNOOPY PROTOCOL

The key to implementing an invalidate protocol using a snoopy scheme is the use of a broadcast medium. Snooping coherence on a bus was first proposed by Goodman [11]. To send invalidation, the core simply acquires the bus and puts the address and invalidation command on the bus. Obviously, if the bus is busy, it acts based on the specified communication protocol. All of the cores are snooping the bus. If the address, which is on the bus, exists in their caches, the appropriate action will be taken; otherwise, they ignore it. In cases where two cores are writing in the same memory reference and they try to put an invalidation message on the bus, serialization of write is important to guarantee program consistency. One implication forced by this serialization is that write to a shared block cannot be completed until the bus access is acquired. Thus, a serialization in block writing permission or serialized access to message medium should be enforced.

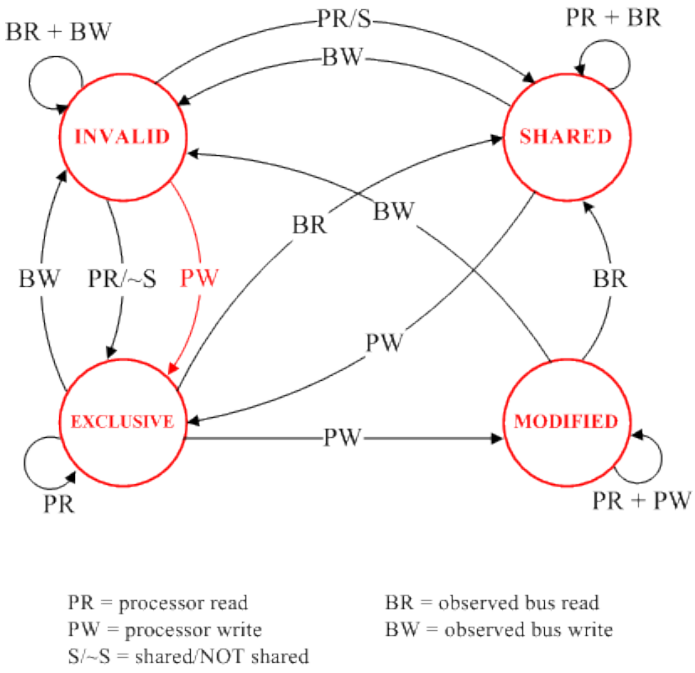

Figure 3- MESI finite machine

The other implication in protocol is finding the most recent copy of an invalidated block on each cache read. The data read operation faces miss because of either invalidation or actual absence. Since the processor is constantly snooping the bus, it easily finds that the miss is because of an invalidation request by another cache, or it should wait for the lower memory level to reply with the data. In the invalidated block case, the processor cancels the request to lower level and waits for the other processors, which have the copy to reply back the correct copy. The additional complexity is between delay time of retrieving data from L3 and private caches of the other processors.

Consequently, invalidation protocols have to be implemented with a write-back scheme at the last level private cache for all processors.

The invalidation is easy to implement. An added bit shows whether the block is dirty (*invalid*) or valid. The other state is when the data is *shared* between the cores; this helps the core to decide about invalidation generation in case of write operation. When the write operation happens to a shared block, the processor changes its state to *modified* state and puts the invalidation command for the associated address on the bus. To avoid unnecessary invalidation messages on the bus, an *exclusive* state shows if only one copy of the data exists. This state is easily added by extra bits to the memory blocks. *Snoopy MESI protocol* is the standard name of this protocol (Figure 3). A further optimization is the addition of *owner* state, which distinguishes the case in which several copies exist and the actual copy in main memory is out of date. This state avoids unnecessary write backs in case there are other attempts to read the same address in the other cores when a core is reading a datum that another core is modifying. This protocol is called *MOESI*. MOESI state transition policy is described in Figure 4.

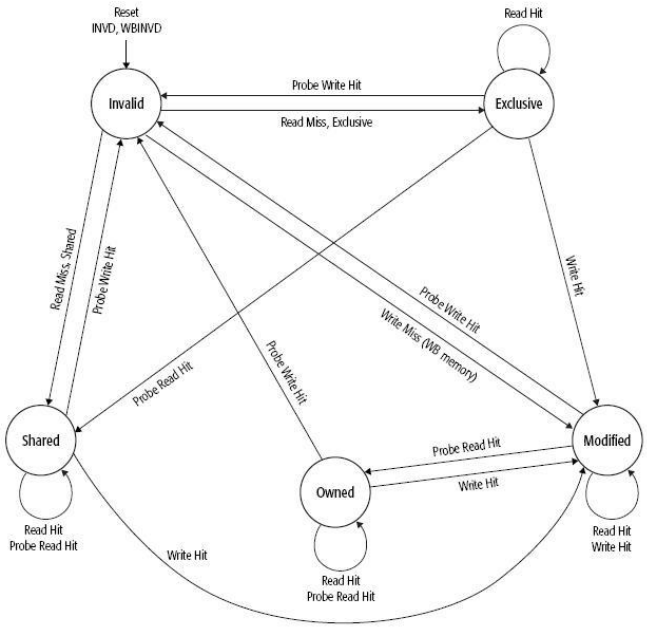

Figure 4- MOESI finite machine

2.2.3 SNOOPY LIMITATIONS AND DIRECTORY SCHEME

As the number of cores grows, or as the memory demand of each core grows, any centralized resource on the chip can be potentially a bottleneck [1]. Even utilizing a high bandwidth bus for current chips could not enable designers to support more than 8 to 10 cores on a chip without experiencing an exponential drop in performance. Bus bandwidth is a bottleneck in snoopy since each coherency miss should be examine in the core. Besides Directory-Based scheme, one approach for a larger number of cores, commonly used in Xeon 700 and core i7, is a directory in the outermost cache (L3). This directory explicitly keeps track of references, which are in the cores of that processor. This method

cannot eliminate the bottleneck due to the shared bus between L3s [24]. This scheme is much simpler than *directory-based*, though.

In *directory-based* cache, a hardware directory unit is added to each processor. This directory keeps the relevant information, such as which cache or sets of caches have copies of the block, whether it is dirty or valid, etc. For implementation purposes, a bit vector in the L3 cache keeps track of which private caches have that specific data block. This approach is used by Intel® QuickPath Technology, which is utilized in core i7 and Xeon [12, 13]. On its own, this solution is not scalable as a DSM system. Scalability implies distributed directory, but in a way that searching for a block does not force broadcasting on a network. The obvious solution is distributed directory along memory, but a restriction is required: for each block, one and only one specific directory is used. This scheme is shown in Figure 5 [24]. More details of this scheme are provided in many architecture books [1, 24].

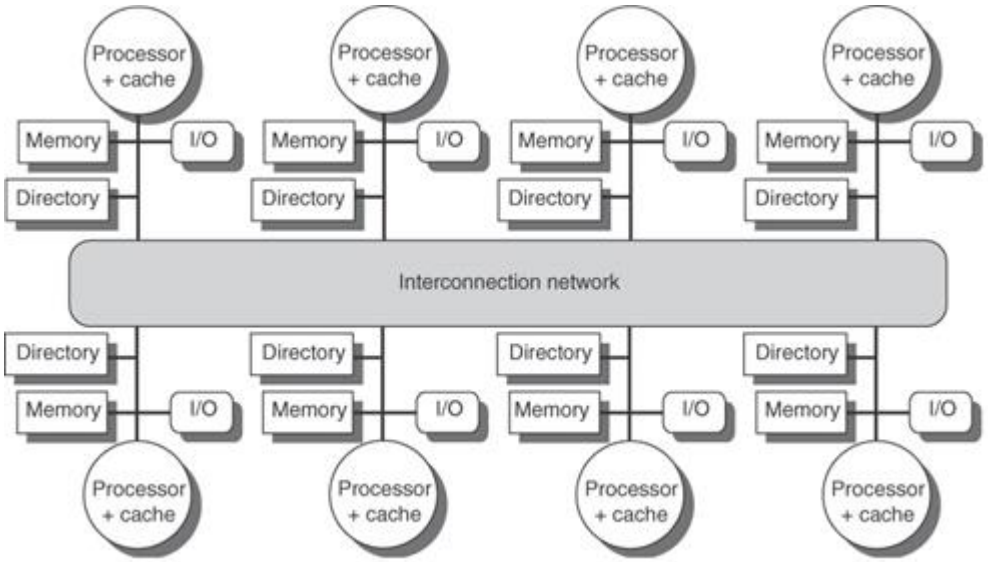

Figure 5- Directory-based scheme

2.3 REVISITING AMDAHL'S LAW AND PERFORMANCE BOTTLENECKS

The performance attributes are more complicated in a shared memory system. One of the most important expected features of parallelism is performance scalability [24]. According to Amdahl's law, the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program. The final speedup formula of Amdahl's law is:

$$S(N) = \frac{1}{(1-P) + (\frac{P}{N})} \qquad \text{Equation 1}$$

Where P is the parallelizable fraction of program and N is the number of processor units [24]. Considering an ideal, fully parallel application, a speedup of two is expected after a 100 percent increase in processors. Each program execution time can be broken into two

portions of parallel and sequential time. Execution of a program shows that even the speedup of isolated parallel portions is not equal to the expected speedup [14]. This means that additional overheads cause this drop in performance.

Several papers study scaling degradation due to overheads. Crovella et al. [15] used a method called lost cycle analysis. They divided overhead into Load Imbalance, Insufficient Parallelism, Synchronization Loss and Communication Loss. All of these overheads are measured in a parallel FORTRAN application on a KSR1 system [16]. However, the communication loss, which is the subject of this paper, is approximated by assigning cycle numbers [16] to misses and hits and calculation by miss and hit rate.

Recently, in a similar paper, Roth et al. [17] categorized overhead into work, delay and distribution. In this classification, communication is under the Hardware Delay category, and since it cannot be measured by profiling software, it is measured as the remaining part of overhead. Roth et al. also tried to use this profiling data to improve performance by algorithm level changes.

Kunz [18] classifies these overheads in three major categories. Equation 2 shows that parallel execution time can be separated into two portions of actual computation and overhead, which prohibits ideal scalability. Regardless of the machine architecture (ccNUMA or SMP), this overhead (sequential and parallel) can be categorized in three different groups of bottlenecks. The first group is overhead caused by the application

itself. Machine architecture is the cause for the second group of overhead, and the interaction between machine and application causes the third group of overhead.

$$ExecTime_N = ParCompTime_N + SeqTime + ParOverhead_N \quad \text{Equation 2}$$

We look at each bottleneck more accurately, and finally focus more on the coherency protocol overhead, as it is the subject of this paper.

2.3.1 APPLICATION-CENTRIC BOTTLENECKS

The characteristic of implemented parallel software is potentially a bottleneck. Although these bottlenecks might be relative to machine specifics such as memory size, processor speed, etc., they can still be recognized as an architecture independent bottleneck. Since these bottlenecks root in high-level algorithm restrictions, a solution is in algorithm-level change or compiler optimization methods. Compiler methods require very comprehensive information, and are not always possible or totally effective. We should consider that some inherent constraints prevent the application from being perfectly scalable. Insufficient parallelism, which leads us to Amdahl's law for multiprocessors, is the most obvious one. By nature, an algorithm might have a sequential part, which is not parallelizable by any means. By implementing an algorithm, a programmer implicitly limits the potential speedup. This is obvious, and a larger data set means a higher level of available parallel resources, impacting the size of serial and parallel portions of the application [18].

As the number of processor units increases, a scalable speedup is desired. However, even solely for parallel portions, the algorithm might not be able to grantee the speedup scalability. Moreover, a level of communication between processors is required. Clearly, the algorithm and programmer determine the ratio of communication to computation. Some other performance bottlenecks are dependence on system call and I/O traffic, which originates in programming and the algorithm method that is used [18].

2.3.2 MACHINE DEPENDENT BOTTLENECKS

Although the communication cost is categorized as an application-centric bottleneck, the execution of a parallel application shows a larger communication cost than the ideal cost expected from the algorithm. The reason behind this additional cost is the characteristics of architecture design. In most cases, performance optimization in this level requires information about some machine-specific features. Since we are looking to shared memory systems, block size can be a good example of reasons behind deviations from ideal communication cost. Memory organization uses a fixed block size to transfer memory references, simplifying access management and taking advantage of locality. Thus, transferring a word is done in the form of a bigger block of words. In a scenario where two processor units need the same block of memory or any resource such as page, etc., while they actually are accessing different sets of words within block memory, *false sharing* happens. False sharing can plummet the performance since it intensifies the communication dramatically [24]. In a less clear case for a programmer, false sharing

happens on an OS level granularity, e.g., a page. This case occurs when disjointed sets of data are organized in the same page. Typically, page-level false sharing occurs on pages that hold thread-private data. The programmer solves the problem by separating unshared data into separate pages or segments. Although the overhead is not completely omitted, it is negligible, especially considering this fact that the programmer benefits from spatial locality. Finding and removing false sharing is difficult to do by compiler. The location of the bottleneck in memory depends on the input size, proposed parallel algorithm, and OS placement decision. A proper memory system can mitigate these bottlenecks.

2.3.3 APPLICATION-ARCHITECTURE HYBRID BOTTLENECKS

This type of bottleneck is due to the interaction between software and architecture. These bottlenecks exist in all multiprocessors, but their effect on performance depends on the cost of the operation in hardware implementation and the overhead of the application interface.

2.3.3.1 THREAD SYNCHRONIZATION

In a multithread program, race condition between threads causes an unexpected and incorrect behavior. Thread synchronization mechanisms eliminate the race condition by managing the access to the shared data. The cost of barrier varies since it depends on the length of parallel sections across threads, and all threads should arrive to barrier before they continue execution. Parallel-section dependencies force an overhead to the

program, since all the treads experience the worst execution time before their arrival to barrier. High level parallel programming such as OpenMP implies barriers. A compiler might be able to check the barriers to make sure if they are necessary; however due to the complexity of such a test on barriers, which requires knowledge and changes in the high level algorithm, other approaches such as *Thread level Speculation (TLS)* are usually taken [19].

Software lock also degrades parallel performance. Wrong lock acquisition might change the behavior of a parallel program to serial. Moreover, lock acquisition by itself depends on the memory system, while the memory system speed in lock acquisition and release affects this bottleneck. However, hardware overheads are often very small compared to long critical section. A compiler might be able to detect non-critical regions inside the locked area and do something about them to shorten the critical section. For infrequent locks, an approach such as synchronization can be taken by the memory system [18].

2.3.3.2 Operating System Bottlenecks

Thread scheduling is an important responsibility of an OS. This task guarantees memory protection and the availability of resources for all processes. Scheduling algorithms decide which thread should have the processor next. As the number of processing units increases, more variables should be taken into account. This means more scheduling overhead. This overhead rises even more in ccNUMA architectures because

scheduling and placement of threads is very critical in ccNUMA system and depends on more variables, such as the size and location of data. There are also bottlenecks associated with the scheduling of multiple multiprocessor scheduling.

2.3.3.3 Communication

In a parallel program, processing units have to communicate with each other for different reasons; this communication forces overhead to the execution time. The cost of this communication varies in time based on the parallel algorithm, data set organization, and architecture. The numbers of utilized cores, scheme (SMP, ccNUMA), and communication backbone (network bandwidth, routing, bus protocol and etc.) affect communication overhead. A specific architecture might cause a hot spot which exacerbates this overhead. Activities that generate communication, such as distributed address space and coherency communication, are good examples of this group. Naturally, the number of communication packets increases as the number of cores and available caches grows. This increase in available resources causes speedup and hides the cost, which is paid for communication. Thus, the ratio of communication to computation is an effective performance metric. As the ratio increases, the scalability of a system speedup diminishes. A higher communication-to-computation level effect is the same as insufficient parallelism.

In an SMP system, messages from the coherency protocol form the majority of network traffic. The focus of this paper is on coherency messages and their effect on expected speedup. In next part, MESI message overhead is studied more accurately.

2.4 COHERENCY PROTOCOL OVERHEAD

In general, cache is used to utilize the locality and reduce the gap between memory and processor unit. This obviously increases the performance of a single-core CPU. As we described at the beginning of this chapter, it is highly coupled with access patterns of the application and machine specifications [20]. Technology leaps in parallelism, multicore, and multiprocessor changed the calculation, though. Although a multiprocessor still takes advantage of locality, an overhead is forced to keep all the caches coherent. This overhead varies from one application to another, but is always visible as deviation from ideal performance. In an SMP system, the amount of messages and data that should be transferred between cores to keep the caches coherent depends on two different issues. First is the executed application memory pattern and data set size. Depending on how many of cores share same data, how often they want to write data, and the order they write and read, the number of required coherency messages and the amount of transferred data are affected. Second, the coherency protocol and memory organization influences the required amount of messages.

In the comparison of a single-core system with a multicore system, a few cases force extra communication and data transfer. Reviewing Figure 3, which shows a MESI

finite state machine, reveals three different cases where extra communication is needed to keep the memory required.

1. *Read an invalidated data*. In this case, the cache line is present in the local cache, but it has been invalidated by another processor. If the coherency issue did not exist, this memory reference request would have been a hit. However, since another processor invalidated this copy, it is a miss. An invalidated copy might be the result of write on another copy, whether this copy was modified, shared, or exclusive. This case might happen in any level of private cache. When a private cache receives a request for a reference and finds it in the cache, it checks whether it is valid or invalidated. If it is invalid, miss is replied to processor, a request for that reference is posted on the bus (or broadcasted on network), and the core waits for updated data. Owner processor(s) reply back to the request for data.
2. *Read miss to shared data*. In a single-core scenario, a miss in private cache is handled by sending a request to a lower level of memory (toward main memory). However, in an SMP scenario, the valid copy of this memory reference might be either in lower memory or the other caches. When memory receives a request for a memory reference that is not present in the cache; it replies back miss and puts a miss message for that memory reference on the bus. As a processor with a valid copy (Modified, Shared, or Owner) receives the message, it put data on the bus.

More issues in this case are solved in implementation, and more are discussed in next chapter.

3. *Write to shared data*. Writing in a cache can be done either in *write back* or *write through.* Most of the memory hierarchies right now use the write back approach. This approach prevents the cache from engaging in the continuous updating of lower shared memory. Consequently, writing into a single core cache loads memory reference into the cache and writes into it, and updating only occurs when the reference is evicted from the private caches. Since cache write usually uses a *write allocation* approach, three conditions might happen:

    i. It is a cache miss and the data is not shared. This is a normal miss case and does not introduce any overhead.

    ii. It is a cache miss and the data is shared. In this case, write allocation approach requires loading memory reference before writing data. This case introduces an overhead in the same manner as read to shared data, which is explained in number two. In other words, the processor does an ld and st micro-operation for one store instruction. As the data is present in the cache, a situation such as cache hit, which is explained in iii, occurs.

    iii. It is a hit and the data is shared. In this case, there is a communication overhead because the processor has to inform the other

processors that this shared data is modified and their copy is no longer valid. This overhead is heavily dependent on architecture, network, and communication protocol.

In this paper, we attempt to quantify the contribution of each of these cases to performance loss in graph benchmarks.

3 RELATED WORKS

3.1 QUANTIFYING SPATIAL LOCALITY

To alleviate the memory wall problem, an increasing fraction of microprocessor chips is devoted to caches. In memory hierarchy, higher levels of cache (closer to the processing unit) are smaller and faster than lower levels (which are farther from the processing unit). This fact leads us to a tendency toward loading data from the L1 cache, which is closest to the processing unit. However, the limited size of caches forces computer architects and programmers to make wiser decisions about where we store the data references and manage memory. Locality plays a major role in this decision making process. Two different terms of locality are *Temporal* and *Spatial* locality.

1. *Temporal locality* means that, if a memory location is accessed, then it is likely to be accessed again in the near future. By looking into the cache access schema of different applications, it can be seen that some memory locations are accessed more frequently than others. Additionally, it can be seen that, for the same memory location, accesses are clustered in time. As a result, when a word is brought into the cache, there is a good likelihood that it will be accessed again before it is evicted.
2. *Spatial locality* refers to the fact that memory locations that are physically near to each other are likely to be accessed nearby in time. Clearly, spatial locality results from the fact that *related* values, such as fields of the same

record or neighbor elements of a matrix, are often stored in close proximity to each other. As a result, when a cache miss causes a memory line to be brought into the cache, there is a good likelihood that words in the line other than the one that caused the miss will be accessed as well.

Historically, the data locality properties of programs have been studied for two different purposes: first, for a better understanding of program behavior (architecture independent behavior); and, second, for utilizing dynamic locality properties for prefetching [21, 22, 23]. The focus in this section is on program locality metrics, which represent the overall memory behavior of program. After gathering detailed statistics about spatial and temporal locality, it can be tempting to make a general decision, or comparisons such as: *application A has more temporal locality than B*. Although this reduction can be useful, it may be an oversimplification that discards information.

Beyond the qualitative descriptions provided in computer architecture books [24], various locality characterization metrics have been proposed in literature. In terms of temporal locality, Pyo et al. [25] introduce *reference distance* as the total number of references between accesses to the same data. Beyls et al. [26] show that this metric cannot exactly predict cache behavior for fully associative caches, but an alternative metric is able to do so. The alternative metric is based on *stack distance*. Mattson et al. studied stack algorithms in cache management and defined the concept of stack distance in 1970 [27]. Applying the same concept as LRU stack distance, *Reuse Distance* is

defined as the number of distinct memory references between two successive references to the same location; reuse distance provides a quantification of the locality present in a data reference trace. This metric has two advantages [28]:

1. LRU replacement policy or its variants are used in most of the caches. This means any distance (d) smaller than cache size (N) is accurately modeled as hit (d<N) and the rest as miss.
2. Reuse distance measures the volume of the intervening data between two accesses and is always bounded by the size of physical data, while other metrics such as time distance can be unbounded.

The major problem of this metric is the high cost of the analysis, which precluded its online use. Neu et al. [28] presented the first parallel algorithm to compute accurate reuse distances by analysis of memory address traces. Reuse distance is used as a metric in Ding and Zhong [29]. The same study [26] uses this locality characterization metric to analyze the distribution of the conflict and capacity misses in the execution of code generated by an EPIC compiler, and reasons on the impact of increasing parallelism in an application on the number of capacity misses.

Weinberg et al. [32] represented temporal locality as a set of reuse distances and corresponding memory operation fraction. The memory operation fraction is calculated by *reuse function* (Equation 3), where *reuse i* denotes the fraction of dynamic memory operations with distance less than or equal to *i*.

An important parameter is to assign weight to the memory references at each reuse distance. For simplicity, this initial study employed a log scale where each memory reference is weighted by the log of its reuse distance with respect to the largest distance considered. Thus, the reuse function provides a single architecture-independent score for each reuse distance [32].

$$f(x) = \frac{\sum_{n=0}^{log(N)} \left((reuse_{2^{i+1}} - reuse_{2^i}) * log_2(N) - i \right)}{log_2(N)} \quad \text{Equation 3}$$

In terms of spatial locality, previous works have attempted to quantify it, mainly via scalar metrics that allow for easy ordering and/or clustering of applications in locality classes [30]. Weinberg el al. [32] defined spatial locality metric by assigning a proportional weight to the strides and ending up with one single score between 0 and 1 (Equation 4). In this formula, $Stride_i$ denotes the fraction of total dynamic memory operations that are of stride length *i*.

$$\sum_{i=1}^{\infty} Stride_i / i \quad \text{Equation 4}$$

We should consider that the spatial locality metric (which constitutes most spatial metrics) is hard to efficiently calculate in run-time [30]. All of the aforementions studies, however, tend to treat the spatial and temporal dimensions of locality as completely orthogonal to each other and thus only offer a pair of uni-dimensional scores. Anghel et

al. [31] propose to generalize these concepts and quantify the entire two-dimensional spatio-temporal locality characteristic of a program accurately.

In this section, we focused primarily on the memory reference patterns of individual processors to their local memory. This means that literatures mentioned above are characterizing only locality through their own caches. Moreover, spatial and temporal localities exist in messages and inter-processor communication [32]. However, even these metrics can provide a good insight into the nature of our benchmarks. Consequently, the first step toward a better understanding of this interconnect locality is looking more accurately at the effect of inter-processor memory communication on machine performance. Then, we see why multiprocessors naturally weaken the existing locality and even force overhead to the machine. Next, cache coherency protocols are discussed, and the reason why inter-processor locality affects machine performance is explained.

3.2 OVERHEAD CHARACTERIZATION

Generally, coherent overhead can be represented by different metrics and measured for different architectures and schemes. In one of the earliest studies, Hennessy et al. [33] reviewed the key developments that led to the creation of distributed cache coherent shared memory. They used a distributed coherent cache prototype (DASH) and compared local and remote access time. They concluded that scaling and speedup are going to be issues in DSM system very soon. In 1980, Emerson et al. [34] tried to characterize parallel applications for the purpose of coherency protocol overhead

estimation. They studied overhead of Barkely Ownership and Firefly protocols by simulating the generated parallel benchmark trace for 11 or 12 processors. The study tried to characterize the synchronization overhead as well as coherency protocol. Since the simulator in this study was not a full system simulator, they assigned a cost to each stage transition as an average cost. Also, memory size of transferred data was not taken into account, since they fixed the block size at 8 words. They finally concluded that write-invalidate protocols have better performance than write-broadcast protocols. The overall calculated wasted cycles were 2% of total cycles.

A number of works characterized the overhead issues on ccNUMA systems. Chanduri and Heinrich proposed a new DSM coherency protocol that does not require *NAK messages* [35]. They measured the detail overhead of NAK messaging in DSM as well as the characterization of lock, synchronization, and total memory operation (without further protocol cycle details). Heinrich et al [36] characterized detailed overhead of parallel application execution on a FLASH multiprocessor, including the overhead of exploited ccNUMA protocol.

Kunz focused on large-scale multiprocessors and characterized and analyzed the execution bottlenecks of a parallel application on a FLASH system [18]. Beside operating system and synchronization overhead, Kunz used several protocol models for the same benchmark and justified the performance tradeoffs. Execution times of different protocols were compared, but the overhead times of the individual protocols were not measured.

Heinrich et al. [37] characterized cache coherency protocols in large scale shared memory processors where the overall performance critically depends on cache coherency protocol. They compared several *NUMA distributed shared memory protocols* and also *Cache-only Memory Architecture* (COMA). Since communication in a large scale multiprocessor is highly affected by network, coherency messaging was observed as the root of major differences between the performances of surveyed protocols. They finally concluded that finding optimal coherency protocol based on overhead data is difficult because a change in machine specifications (for example, cache size) can change the overhead of protocols even for fixed applications.

Molka et al. [38] presented fundamental details on ccNUMA architecture on Intel Nehalem microarchitecture and its memory controller, Intel® QuickPath Interconnect. In this paper, bandwidth and latency between different locations is profiled, including on-chip cache latency and off-chip ccNUMA memory latency. This work did not characterize benchmark memory behavior or detail protocol overhead, but measured their effects on latency and bandwidth. With a similar approach, Peng et al. compared AMD Athlon64 and Intel Core 2 Duo [39]. They profiled the execution of STREAM and STREAM2 on an actual machine and recorded bandwidth and latency of memory access, but did not provide details on protocol performance and timing issues.

Kumar and Huggahalli examined a specific kind of coherency protocols in a network traffic analyzer machine, which uses general-purpose processor containing

Intel® Core™ micro-architecture based processors [12]. They measured the wasted time in various protocols for coherent write and read between *NIC* (*Network Interface Chip*) and processor caches, using *DCA* (*Direct Cache Access*). They did not take cache coherency through cores into account, but they showed that for this specific coherency application, the system is mostly stressed by coherency overhead in 10 Gb/s bandwidth [40].

Montaner et al. [41] decoupled the remote memory needed for computation from remote memory needed, due to limited space in a certain node in clusters. In this approach, OS hot-plugged support is required, and OS should be aware of free space in other nodes. Execution time was then measured, which showed how remote data transfer can cost overhead to the system.

Borroso et al. characterized the performance of a specific memory system facing commercial workloads [42]. They looked at the performance of two commercial workloads, online transaction processing (OLTP) and decision support systems (DSS), on an alpha multiprocessor using both simulation and monitoring actual system. Their emphasis is primarily on the characterization of only those different aspects that are needed to see the trend of performance on different specification. Miss rate, sharing patterns, etc. in various cache sizes are studied. Borroso et al. mostly tried to characterize memory misses in detail such as false sharing, true sharing, replacement and cold misses.

Foglia et al investigate the performance of MIPS-based 4- and 6-processor systems using MESI coherency protocol facing Electronic Commerce applications [43, 44]. They looked at false sharing and true sharing and portion of misses as well as invalidation miss percentage. Since they used a trace-based simulator [45], they could not provide any timing data regarding execution time overhead.

The human conventional wisdom as well as results from these papers says that communication overhead is highly correlated with the specific application and its data-intensiveness. The focus of this paper is on data-intensive graph applications. So, to continue, we talk more about graph applications and their characteristics.

3.3 PERFORMANCE AND OVERHEAD OF GRAPH APPLICATIONS

Vetter et al. [46] showed that extreme-scale applications have specific characteristics, some of which are expected and some of which contradict conventional wisdom. Their metrics are classified into two classes, computation and communication. Most of these metrics are measured for 12 HPC benchmarks including Graph500. Reuse distance and memory bandwidth as well as communication patterns are metrics which are studied. However, this paper does not provide complete data for graph benchmarks such as Graph500 or SSCA2, which are known for low locality.

Checconi et al. [47] focused on reducing the scaling overhead of Graph500 on a NUMA machine by optimizing the algorithm. Using several optimization techniques,

especially those for mapping virtual processors to nodes, the BFS algorithm showed a better scalability, although the overheads were not measured independently.

The execution of Graph500 on an 82-node cluster and the measurement of traversed edges per second (TEPS) as a performance measurement by Angel et al. [48] demonstrated Graph500 as a benchmark to measure a computer's ability to efficiently access memory. Cui et al. [49] and Yasui et al. [50] showed that in the OpenMP/MPI hybrid model for multi-node and the OpenMP model for single node execution of Graph500, communication is the chief influence on performance (GTEPS), and the optimization of memory transfer can increase performance by 50 percent.

3.4 SUMMARY AND RESEARCH CONTRIBUTION

As we saw earlier, scalability is an expected feature in a parallel system, but various overheads reduce the ideal speedup as the number of processor units increases. Previous studies show that communication between cores is a very substantial factor in this speedup limitation. In a single core machine, cache hierarchy is a solution to alleviate memory wall. However, in a multicore system, these caches have to communicate with each other to keep the shared data coherent.

Locality of data is a substantial factor in the amount of required communication. Spatial and Temporal locality are hard to quantify, and it gets worse when it comes to multicore processors. However, studies which are discussed at the end of this chapter showed that graph applications have a very low level of locality. In graph applications,

less locality leads processor units to more communication and more cache coherency overhead. Since most of the papers showed that such an overhead is large but did not quantify it, we quantified this overhead and investigated how it affects scalability.

Since this kind of data is not available through software profiling, we have to use a cycle accurate simulator. This simulator has to be close to an actual machine to provide data, which can then be validated by data from a real-world machine. The methodology of our experiment is described in the next chapter, which also discusses the machine specifications, simulator specifications, how overhead is measured, and practical points about protocol. After measuring the number of cycles that coherency protocol wastes, we can calculate how much performance loss is directly correlated to coherency protocol.

4 METHODOLOGY

4.1 QUALIFYING BASELINE PERFORMANCE AND OVERHEAD

As we discussed previously, parallelization overhead appears when we increase the number of computation units. When this occurs, the execution time is not affected as much as the *scaling factor*. In this work, the definition of *overhead* is the deviation of execution time from ideal execution time. However, we should take into account that not all of the execution time can be diminished as scaling factor rules. According to Equation 5, execution time is divided into two parts: parallel and sequential computation times. *Amdahl's law* says the sequential part is not affected by scaling factor as we increase the number of parallel processing units. This sequential time limits the ideal speedup.

$$ExecTime_N = ParCompTime_N + SeqTime \qquad \text{Equation 5}$$

Consequently, we should know what fraction of a program is solely sequential and what fraction is parallelizable for a specific number of parallel processing units. This sequential time is measured for each individual benchmark on a real-world platform [14, 51]. Now, our achievable speedup is less than the number of parallel processing units as we took into account that a fraction of the application is naturally *non-parallelizable*. This ideal speedup varies based on the parallelizable fraction of the program as it is also bounded by scaling factor.

$$ParExecTime_N = ParCompTime_N + ParOverhead_N \qquad \text{Equation 6}$$

The gap between the execution of the benchmark on the machine and the ideal execution time of the parallel program is what we call *parallelization overhead* (Equation 6). Looking at Equation 7 and section 2.3, this overhead can be broken into different components.

$$ParOverhead_N = LibOverhead_N + BarrierTime_N + LockCSTime_N + CoherencyTime_N ExecTime_N \quad \text{Equation 7}$$

In the previous section, we reviewed the direct measurement of each component in literature and showed that coherency overhead has been never measured for recent multicore processors, especially not directly and cycle-accurately. Using tools such as the PGOMP binary instrumentation tool [51] enables us to measure all of the overhead components directly, except the coherency time [14]. Consequently, one approach is to use the data collected by PGOMP to find all overhead portions except coherency overhead; the remaining part has to be coherency overhead [14].

This work collects coherency protocol data directly. To directly measure the coherency overhead time, we need to access some additional underlying information beside the miss or hit rate, which are available through profiling tools. An alternative for collecting such data is using a cycle-accurate simulator. This cycle-accurate simulator has to be a full system simulator because we need to simulate the multicore processing unit,

interconnect and communication between cores, and memory hierarchy. The advantage of a full system simulator instead of only memory system simulator is a guaranteed match between real-world machine and simulated machine. A few problems are standing in the way of finding and using a cycle-accurate simulator:

1. This simulator should simulate an existing architecture. Due to the extremely large development time of a simulator, not enough cycle-accurate simulators are available. Most cycle-accurate simulators are old, outdated, or obsolete. From the few cycle-accurate simulators, we need a simulator to simulate our existing x86 real-world machine. Existing x86 simulators are Superscalar, Zesto, PTLsim, Gem5, and MARSSx86 [52]. Superscalar [53], Zesto [54], and PTLsim [55] are all single-core simulators. Gem5 [56] is a full system, which has limited support for x86 but does not support MMX or SSE instructions. MARSSx86 is an x86 cycle-accurate simulator [57, 58]. It is a full system simulator and contains a cycle-accurate model for interconnect, memory controller, and memory hierarchy as well as multicore processor model. MARSSx86 uses PTLsim as a core model. This simulator suits our needs better than the other options.
2. Validation and accuracy is a questionable issue. Since academic research usually studies relative performance improvement rather than the imitation of existing real-world systems, academic simulators are rarely validated against real-world machines. Even when they are validated, a huge error is shown. MARSSx86 is

one of the few simulators that are validated against actual machine. Simulation error varies among different metrics from 0 percent to 100 percent. We talk about validation in more detail in section 4.2.

3. The biggest problem of cycle-accurate simulators is extremely long simulation time. While a real processor has a performance in the order of thousands of *MIPS*, a simulator has performance of one to hundreds of *KIPS*. The long simulation time issue is a critical issue, especially when a multicore simulator is used. This long simulation time may make the full system simulation impossible when the number of cores increases. MARSSx86 simulates about 160 KIPS for a single core [57]. This speed critically limits the size of application that we can simulate, so the benchmark should be selected wisely. Benchmark selection is discussed in section 4.5.

4.2 SYSTEM VS SIMULATOR SPECIFICATION

An Intel Xeon E7 2860 machine is used as the real-world platform. It has 10 cores on chip, which share a 24MB *shared L3* cache, while each core has two levels of private caches. Detailed specifications are shown in Table 1. MARSSx86 is developed, which is very flexible to change configuration. We wrote our configuration to imitate Intel Xeon E7 2860. The configuration file is available in Appendix A. More information about memory hierarchy, such as latency, interconnect, and protocol of these two platforms is provided in section 4.3.2.2 and Table 3.

Processor	Intel Xeon E7 2800	MARSSx86
Code Name	Westmere-EX	-
Cores	10	10
Clock Rate	2.26 GHz	200KHz
Max Ins Decode per Cycle	4	4
ROB Size	128	128
Int ALU	3	3
FP Units	3	3
LSQ Size	96	96
L1D Cache	32KB per core 64 byte lines 8-way set associative	32KB per core 64 byte lines 8-way set associative
L1I Cache	32KB per core 64 byte lines 4-way set associative	32KB per core 64 byte lines 4-way set associative
L2 Unified Cache	256KB per core 64 byte lines 8-way set associative	256KB per core 64 byte lines 8-way set associative
L3 Unified Cache	24MB shared 64 byte lines 24-way set associativity	24MB shared 64 byte lines 24-way set associativity

Table 1- Real-world platform specification

MARSS community validated the simulator against Intel Xeon 5620 with 4 cores for benchmarks such as SPEC, STREAM and PARSEC (Table 2) [57, 58]. The IPC metric showed less than 2 percent variation for all the reported benchmarks. Caches and main memory round trip latency had a 1 percent error. CPU function unit latency was up to 91 percent different from the real-world machine.

Table 2- Difference between real-world and simulated Intel XEON 5620

Metric	Maximum Difference
IPC	200 %
Cache Round Trip Latency	1 %
Function unit latency	91 %
IPC variance among different pthread configurations	0.4 %

Since most differences are not very large, especially the difference in memory latency l, using MARSSx86 looks reasonable. In the result section, we will report our validation data. We continue with a brief explanation of the MARSSx86 structure, with focus on memory hierarchy.

4.3 MARSSX86 STRUCTURE

MARSSx86 is a tool for cycle-accurate full system simulation of the x86-64 architectures, specifically multicore implementations. By using QEMU, MARSSx86 provides a full system emulation environment with a model for chipset and peripheral. The detailed simulation of x86-64 ISA, which consists of detailed pipeline model, is done

through PTLsim. It also has a detailed model for coherent cache and on-chip interconnections. Both write-back and write-through schemes in caches at any level are supported. As coherency protocol options, MESI and MOESI are available. As an interconnect model, point-to-point, split-phase, on-chip bus, and the switch interconnect model are available. Also, it has a simple DRAM model that simulates bank conflicts and DMA channels [57, 58]. In Figure 6 [57], the general schema of simulation and interaction between simulator and emulator is shown. All the information provided in this section can be found in the PTLsim manual [55] and MARSSx86 documents [57, 58].

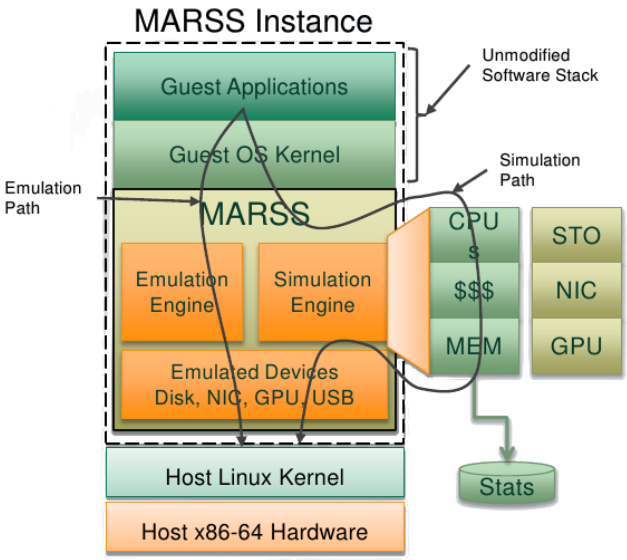

Figure 6- MARSS general schema

4.3.1 CORE MODEL

As a core model, MARSS uses PTLsim. PTLsim supports the full x86 instruction set with all extensions, including x86-64, SSE/SSE2/SSE3, MMX, and x87. Figure 7

shows the high level overview of PTLsim [52]. As PTLsim receives x86 instructions, it converts them to RISC-like instructions called *uops* (micro-operationa). This is the same as processors' uop-translation stage, where each instruction is broken to one to four uops. Each of the 129 uops of PTLsim has three source registers and one destination register. Instantly after making uops, PTLsim pre-decodes them and call them *transops*. The only purpose of this early decode is faster simulation. Then, a group of transops make a *basic block* (BB). Each basic block is indexed by a structure, which includes virtual address and physical page frame number of first transop. In continuance, transop and uop can be used interchangeably. Each BB consists of 64 uops and terminates by condition operation or barrier operation. Then uops go through a pipeline, which has *fetch*, *rename*, *dispatch*, *issue*, *complete*, *writeback*, and *commit* stages.

In the fetch stage, the simulator directly fetches pre-decoded micro-operations from the basic blocks, but simulates the Icache by probing the cache based on current virtual address. It probes the Icache and checks whether the current address of basic block is in the cache or a miss has happened. If a miss has happened, the simulator stalls the fetching. A branch predictor is used in this stage to predict the next uop when the simulator faces branch operation.

The renaming stage starts by reading a configurable number of uops from the fetch buffer. The registers of these uops are mapped into two register files, *Physical Register File* (PRF) and a *Retirement Register File* (RRF). Then, these uops are placed in

a *reorder buffer* (ROB) and wait for the next stage. The number of cycles for renaming stages and the number of ROB entries are configurable.

During the dispatch stage, the simulator dispatches a configurable number of uops from ROB into scheduling queues. There are four scheduling queues: three integer queues and one floating-point queue. The first integer queue serves an ALU and an ALUC, which handles multiplication and division as well as conditional operations. The second and third integer scheduling queues serve an ALU and an LSU which handles load/store operations. The floating-point queue is handled by several floating-point computation units.

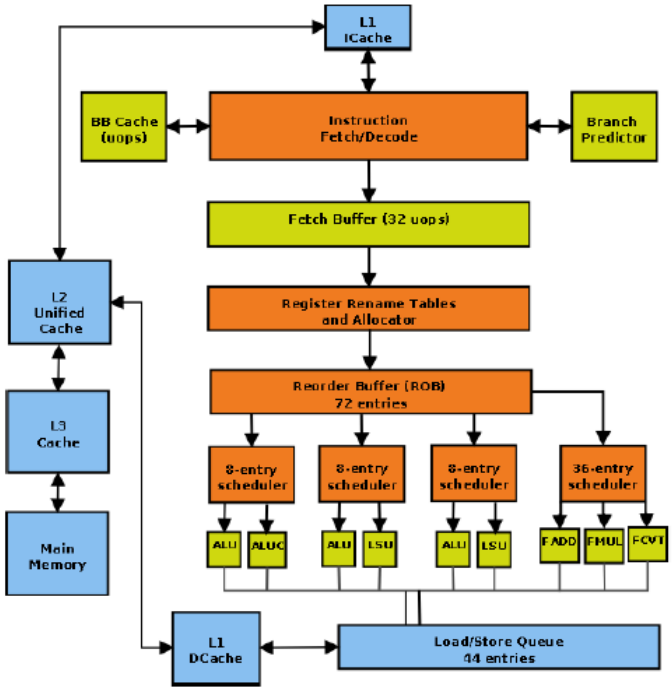

Figure 7- Micro architecture of PTLsim Out-of-Order core

In the issue stage, the oldest *ready* entry of each queue is issued to execution units. The maximum of one entry is issued and released from queue per cycle. A configurable delay rules how many cycles an uop stays in the execution unit. As you can see in 4.3.2, store uops become ready as soon as their addresses are computed, and may be issued before their stored data is available while the other uops become ready when all of their input operands are ready. In the complete stage, the simulator marks the uops, completing execution, and puts them on the forward-bus to send the results for the uops which are waiting for them. Then, in the writeback stage, PTLsim writes back the results into the *physical register file* (PRF). Finally, the simulator retires the uop and puts it into the *retirement register buffer* (RRB). An uop only commits when all the uops belonging to an instruction are completed and ready to commit. This guarantees atomicity of x86 instructions.

The Westmere pipeline is illustrated in Figure 8 [59, 60]. MARSSx86 cores are capable of being configured as close approximations Westmere cores. Although the core configuration is not our concern, the core configuration is available in Appendix B. Since we are simulating Westmere-EX, we are connecting 10 cores in the processor.

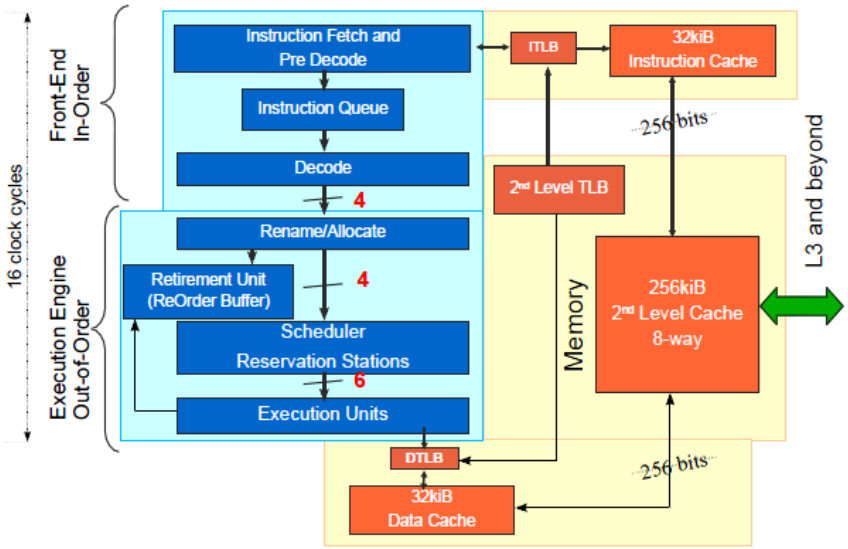

Figure 8- Core Schema in Westmere

4.3.2 LOAD/STORE IN SIMULATOR VS. REALITY

MARSSx86 tries to imitate real-world x86 architectures. Since our concerns are only how the load and store operations are issued and how they are handled by memory hierarchy, we can compare real hardware and the simulator in two parts: first, the procedure of issue stage for load/store and the sending request at the processor side; and, second, memory hierarchy and protocol implementation at memory side.

4.3.2.1 Load/store handling in processor

Load and store are issued by two special methods to handle their various dependencies. These dependencies can occur because some loads or stores may be overlapping. For example, in the PTLsim out of order model, a given store may merge its

data with a previous store in the program order. First, the physical address for load/store is generated by looking at TLB. TLB is also simulated and it may face miss. The TLB miss penalty is set by configurable delay. In this stage, some exceptions, such as page fault, are handled. After the generation of physical address, and the check for exceptions, the simulator checks the *load store queue* (LSQ) backward in time to the head of LSQ for any possible dependency. This ensures that a simulator that loads, and which may need to forward data from a store (*store forwarding*) always references exactly one store queue entry, rather than having to merge data from multiple smaller prior stores to cover the entire byte range being loaded. *Store forwarding* (SF) is implemented in Westmere as well as MARSSx86. SF means when a load data follows a store that reloads the data which is just written by the store into the memory, the micro-architecture can forward the data directly from the store to the load in many cases [59, 60]. This only happens after checking that:

1. The store must be the last store to that address prior to the load;
2. The size of the store must be equal to or greater than the size of data being loaded; and
3. The load data must be completely contained in the preceding store.

The other important practical issue is *memory disambiguation*. A load instruction micro-op may depend on a preceding store. Many micro-architectures block loads until all preceding store address are known. The memory disambiguator predicts which loads will

not depend on any previous stores. When the disambiguator predicts that a load does not have such a dependency, the load takes its data from the L1 data cache. Eventually, the prediction is verified. If an actual conflict is detected, the load and all succeeding instructions are re-executed.

The current version of MARSSx86 always uses write-allocation [59, 60]. However, Intel64 architecture uses write-allocation scheme for most of the cases. This architecture does not write-allocate on a write miss when the write operation is *non-temporal*. When data is *streamed* in and out of the processor, as the case may be with SIMD/vector operands, there is no need to store this data in the cache as it is not expected to be needed in the near future. Write allocate operations are costly, especially when the block slated to enter the cache would have to evict a modified block already resident in a conflicting cache location. The Intel compilers may select the non-temporal write operations when it is evident that the written data are streamed out of the processor.

Finally, after handling exceptions and dependencies, the simulator sends the request to the memory hierarchy. Connections between cores, caches and main memory are configurable. Switch, bus, and split-phase bus are our options. In continuance, we look at the interconnect and memory hierarchy of the simulator and compare them to Westmere-EX.

4.3.2.2 Memory Hierarchy in MARSSx86 vs. Westmere-EX

In Westmere architecture, each core has its own L1 and L2 private cache. Also Icache and Dcahce are separated from each other and have different associativity. Cache hierarchy in MARSSx86 has this flexibility to be configured, same as our real architecture. Table 3 shows different features of both memory hierarchies. The size and associativity of the simulator are set and completely match the real-world platform. Still, some differences are inescapable.

First, the currently available MARSSx86 version supports only inclusive cache hierarchy. In Westmere architecture, the L3 cache is inclusive (each memory reference in L1 and L2 has to be present in L3 as well) but L2 is non-inclusive (data references in L1 are not necessarily present in L2). This difference is negligible because it is only the case for L2. Cache latency in Westmere is provided from [59, 60].

Table 3– Memory hierarchy in Intel Xeon E7 2800 and MARSS

Processor	Intel Xeon E7 2800	MARSSx86
L1D Private Cache	32KB per core 64 byte lines 8-way set associative Latency: 3	32KB per core 64 byte lines 8-way set associative Latency: 3

L1I Private Cache	32KB per core 64 byte lines 4-way set associative Latency: 4	32KB per core 64 byte lines 4-way set associative Latency: 4
L2 Unified Cache	256KB per core 64 byte lines 8-way set associative MESIF Protocol None-inclusive Latency: 10-12	256KB per core 64 byte lines 8-way set associative MESI Protocol Inclusive Latency: 12
L3 Unified Cache	24MB shared 64 byte lines 24-way set associativity Writeback Inclusive Latency: 36-40+	24MB shared 64 byte lines 24-way set associativity Writeback Inclusive Latency: 36

Second, the MARSSx86 interconnect and Westmere are not completely matched. Point-to-point, switch, bus, and split-phase bus are supported in MARSSx86. Point-to-point does not have a queue, and it connects two components without any delay. This connection is used for the connection between private caches and cores. As the core schema shows in Figure 8, the connection between core, L1, and L2 can be simulated as point-to-point without any delay since the delay is already reflected in cache delay.

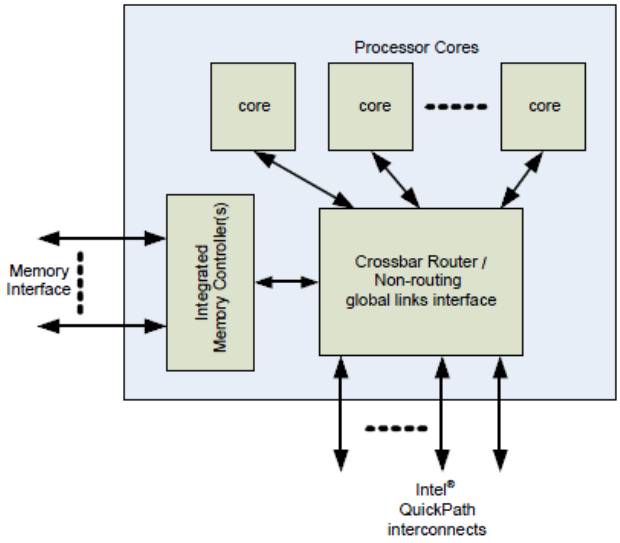

Figure 9- Core interconnects

Figure 9 [12] illustrates the *un-core* interconnects in Westmere. In this processor, the un-core domain is essentially a shared *last level L3 cache* (LLC), a memory access chip-set (*Northbridge*), and a QPI socket interconnection interface [59, 60]. Cache line requests from the on-chip ten cores, from a remote chip, or from the I/O hub are handled by the *Global Queue* (GQ), which resides in the un-core (Figure 10 [60]). The GQ buffers, schedules, and manages the flow of data traffic through the un-core. A *cross-bar switch* assists GQ in exchanging data among the connected parts [59]. The operations of the GQ are critical for the efficient exchange of data within and among Westmere processor chips. The GQ contains 3 request queues for the different request types:

- Write Queue, a queue for store memory access operations from the local cores;
- Load Queue, a queue for load memory requests by the local cores; and

- QPI Queue (QQ), a queue for on-chip requests delivered by the QPI links.

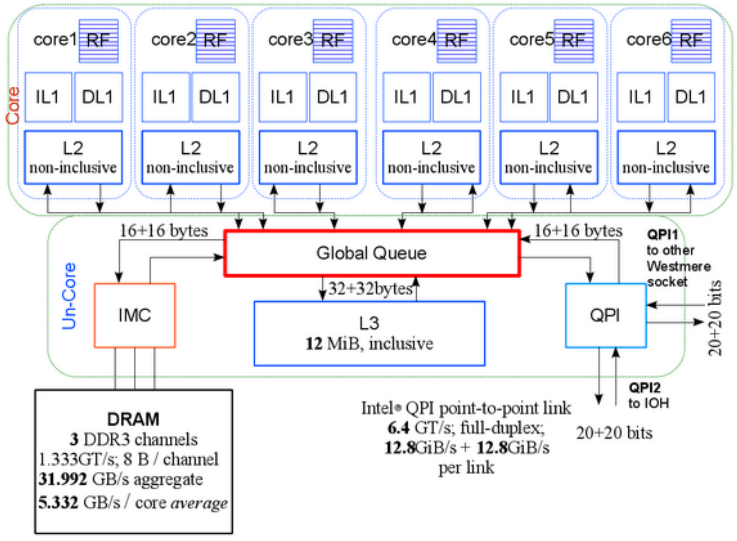

Figure 10- Westmere-EP (6-cores) on-chip interconnect

Westmere-EX has 4 Intel® QuickPath Interconnect (QPI) terminals, which can provide off-chip connection (Figure 11 [60]). QPI provides a fast point-to-point interconnect between chips, as illustrated in Figure 12 [60]. QPI provides a backbone and network protocols such as network layer abstractions and message packaging policy for efficient connection. The interconnect link pair operates two unidirectional links simultaneously, which gives a final theoretical raw bandwidth of 25.6 GB/s.

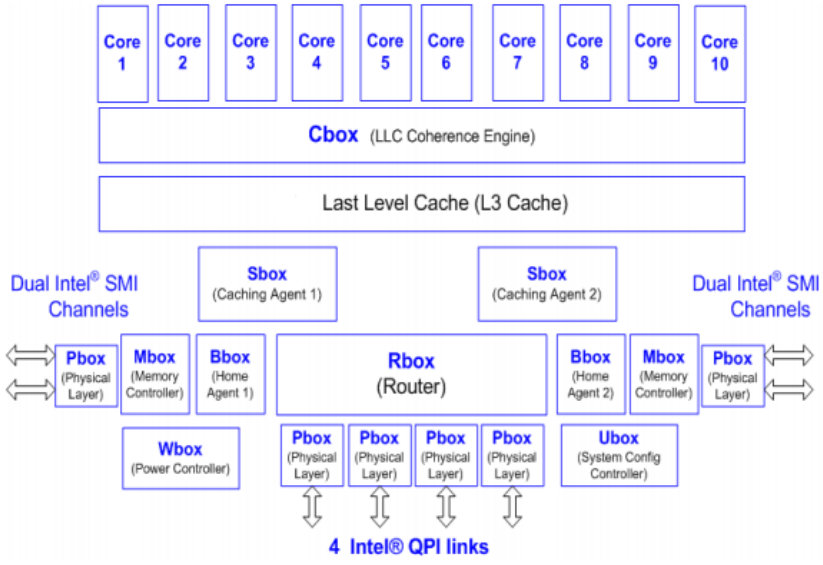

Figure 11- Westmere core off-chip interconnects

In MARSSx86, Global Queue can be simulated by either split-phase bus or switch. It is important to note that in either of these network solutions, coherency protocol connection is not affecting data bandwidth. Also, QPI channels between processors can be simulated in MARSSx86 theoretically. However, since we did not use the second processor of our real-world platform and computation was kept on-chip, we do not need to simulate them.

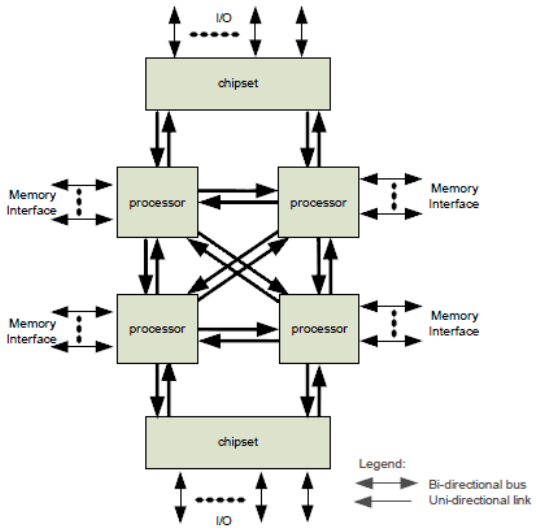

Figure 12- QPI point-to-point connections

The third issue is coherency protocol. Westmere is using snoopy protocol with a small directory part. This is same as what we explained in sections 2.2.2 and 2.2.3. QPI also provides a platform for processors to be able to snoop each other, and provides distributed directory-based protocol [12]. Westmere uses MESIF, which has one state more than regular MESI. A specific cache memory is chosen to store a shared block in the *Forward* state, and is allowed to forward it to other requesters. The presence of this state prevents the collision of data sent on the bus. On a switch network, it ensures that unnecessary packets are not sent to the data requester. However, MARSSx86 only provides snoopy MESI. The configured protocol in simulation is close to Westmere, but the controller is slightly different since the limited directory is not implemented.

Fourth, MESI protocol faces two practical difficulties. First, the atomicity of multistep miss process should be guaranteed [24]. Steps of an upgrade miss are miss detection and invalidation message generation, access to transmission media, and processing of the invalidation in the other processors. In a scenario where two processors try to update a cache line, this atomicity is crucial. This atomicity can be solved if the implemented platform guarantees delivery of the message as the processor accesses the network. Both the winner processor and the loser processor understand the situation, and stop or continue the process. This atomicity and guaranteed order is easily implemented in MARSSx86, since MARSSx86 uses a signal-event-callback approach. A second difficulty occurs in a write-back cache, where the data for a read or write miss can come either from memory or from one of the processor caches, but the requesting processor will not know a priori where the data will come from [24]. In most bus-based systems, a single global signal is used to indicate whether any processor has the exclusive (and hence the most up-to-date) copy; otherwise, the memory responds. These schemes can work with a pipelined interconnection by requiring that processors signal whether they have the exclusive copy within a fixed number of cycles after the miss is broadcast. This problem is also easily handled in simulation, since, when a cache or memory answers a request in the same cycle, the simulator checks all entries in the queue, and the other answers, which are not from the critical path (answers from coherent memory with higher delay), will be annulled.

4.4 COHERENCY OVERHEAD MEASUREMENTS

For the purpose of coherency overhead measurement, we tracked x86 instructions from the top of the pipeline. Since the uop structure also has the information about the belonging instruction, it is possible to profile the application by size or category of instruction in the issue stage. We start tracking each instruction from load/store issue stage in the core model. As the processor sends the request to cache hierarchy, we check if it is one of the three cases, which we explained in section 2.4 as coherency overhead. By editing the request structure, we are able to track the request deep into the cache hierarchy. We also track the source and the size of the instruction to the end.

When the cache controller receives the request, it starts to process the request. We are interested in three special cases, as we explained in section 2.4. In the first and second cases, which are *Read an invalidated data* and *Read miss to shared data*, we look at the state machine and mark the request when they happen. Due to the signal-even-callback nature of simulator, we finalize our measurement in the callback function of the data arrival signal. When the data which belong to our target request arrive into the processor, that signal will be triggered. We can collect the number of wasted cycles based on the size of requested memory as well. The other information is the sharing and communication pattern. This helps us to know what fraction of shared data is provided or invalidated by each core.

The third case is *write to shared data.* Section 2.4 explained this case in detail and classified it in three categories. If it is a cache miss and the data is not shared, the request is not a coherency overhead. If it is a cache miss and the data is shared, since Westmere is using write allocate scheme, the wasted cycles are calculated in the *read miss to shared data* case because the simulator first brings the data to the cache and does that by handling the store uop as a load uop first. If it is a hit and the data is shared, the only overhead is snoopy overhead because no data is needed to transfer. As the simulated and real-world interconnect networks are split-phase, snoopy messages and data messages are not using the same media and consequently are not affecting each other's bandwidth. In this case, the delay of updating status in lower memory hierarchy is measured.

Finally, the time, which is measured as coherency overhead, is used for further calculations based on what we explained in 4.1. The data that is provided from the real-world machine is used to see how much speedup is lost in scaling, and then the measured coherency overhead is used to see what fraction of it is due to the coherency overhead.

4.5 BENCHMARKS UNDER EXAMINATION

As we explained in section1.1 and 3.4, our focus is on parallel graph applications. As is shown in section 3.3, Graph applications have very low levels of locality and have many load/store operations. The memory footprints of graph applications are usually

large and random. All of the examined benchmarks use OpenMP for parallelization. Three different benchmarks are used to measure coherency overhead:

1. *Graph500* [61]: This benchmark is a compact application that has multiple analysis techniques accessing a single data structure, which represents a weighted, undirected graph. This benchmark includes a scalable data generator, which randomly generates tuples containing the start vertex and end vertex of each edge. The second kernel performs a breadth-first search (BFS) over the graph. Overall, the benchmark goes through six steps:
    A. Generate the edge list. The scale factor determines the scale of data set. We used a scale of 16 for this benchmark.
    B. Construct a graph from the edge list (kernel 1). The vertex numbers are randomized, and a random ordering of tuples is presented to kernel 1. The generated list does not exhibit any locality to be exploited by computation kernels.
    C. Randomly sample 64 unique search keys.
    D. Search and compute the parent array (kernel 2). The performance of this kernel reflects (*i*) architecture throughput when executing concurrent threads, (*ii*) resilience to hot-spotting when many of the memory references are to the same location, (*iii*) the efficiency when each thread is

asynchronous side effect of others, and (*iv*) the effect of dynamic load balance unpredictability.

E. Validate that the discovered array is correct.

F. Compute performance information.

2. *SSCA2* [62]: This is a graph theory benchmark, which represents computational kernels of biology, complex network analysis, and national security. This benchmark is based on *the HPCS Scalable Synthetic Compact Applications graph* analysis benchmark. SSCA2 is characterized by integer operations, a large memory footprint, and irregular memory access patterns. It has multiple kernels accessing a single data structure representing a weighted, directed multigraph. In addition to a kernel to construct the graph from the input tuple list, there are three additional computational kernels to operate on the graph. Each of the kernels requires irregular access to the graph's data structure, and it is possible that no single data layout will be optimal for all four computational kernels. We used a scale of 9 for this benchmark.

3. *HeatedPaled* [63]: This algorithm is one of the regular benchmarks of parallel applications. This code solves the steady state heat equation on a rectangular region. The region is covered with a grid of M by N nodes, and an N by N array is used to record the temperature. Every iteration, it solves the heat equation for all elements and calculates the difference in average temperature between four

corners of the grid. If the difference is less than an epsilon, it is in steady state and the calculation finishes. We used a 300 by 300 grid for this benchmark.

4.6 SUMMARY

In this chapter, we started with our definition of overhead, parallelization overhead and coherency overhead. We explained the detail of our simulated and real-world platforms. We compared details of them in both processor side and memory hierarchy side. The obligatory and configurable differences and matches were depicted. At the end, the data collection and overhead calculation methodology were cleared and the examined benchmarks briefly described. In next chapter, we show the results of this measurement and calculation and show how coherency affects the scaling and speedup.

5 RESULTS

This chapters starts with the measurement of a spatial locality metric that we explained in 3.1 to show that graph applications have low locality. In section 5.2 the validation results from the execution of Graph500, SSCA2 and HeatPlate are shown, and. The effect of simulator accuracy on final result is also discussed although in section 4.3, the validation results of MARSSx86 for PARSEC and SPEC benchmark are presented. Then, in 5.3, the barrier, lock, and coherency protocol overheads of benchmarks are measured by an instrumentation tool. In 5.4, 5.6, the coherency protocol overhead is directly measured and discussed.

5.1 LOCALITY CHARACTERIZATION

According to the discussion made in Section 3.1, we characterize the locality of our benchmarks using the spatial locality metric formula that is provided in Equation 4. This metric sums all of the strides by their size and outputs a number between zero and one. This locality metric is not an accurate measurement for locality; however, it can give us a good insight about how the graph benchmarks (SSCA2 and Graph500) are different from regular parallel applications.

This locality metric is reported by Weinberg et al. [32] for several HPC benchmarks. In Figure 13, this spatial locality metric is illustrated for FFT, HPL, GUPS and STREAM. We measured this metric for three benchmarks that are used in this paper, and which are shown in Figure 13 as well. As we see, Graph500 has the lowest spatial

locality score. This confirms our claim that graph applications usually have very poor locality. On the other hand, HeatPlate, as a normal parallel application, has an acceptable locality.

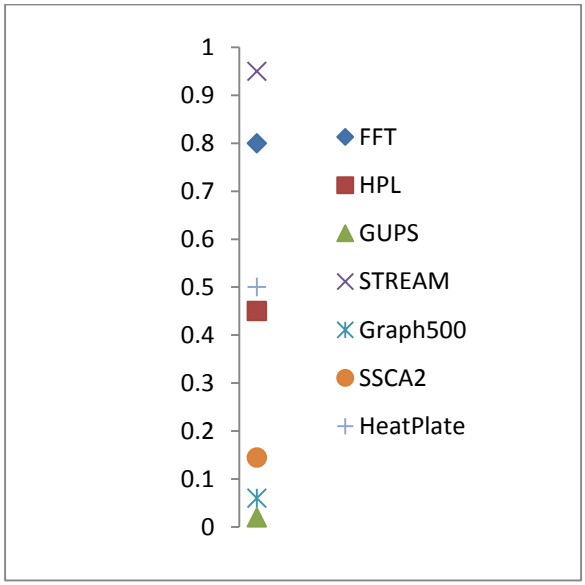

Figure 13- Spatial locality metric

5.2 VALIDATION AND SIMULATOR ACCURACY

The simulator is configured to imitate a Westmere-EX machine. However, since the detailed specifications of the CPU are not provided by vendors, a variable error is introduced to the system. This error varies as we change the number of threads and the benchmark, because we utilize a different number of cores and execute a different distribution of instructions. Since both pipeline and memory hierarchy (caches and interconnects) are modeled in a full simulator, the accuracy of both parts should be evaluated. It should be mentioned that these two errors are highly correlated. The

simulator is expected to have an execution time very close to the execution time of the actual machine. However, a combination of pipeline error and memory subsystem inaccuracy causes a variable error in execution time. A big source of pipeline inaccuracy is uop translation. As we explained in Section 4.3 each x86 instruction is broken into uops before entering into the pipeline. Since vendors do not provide detailed information about uops and the instruction translation process, the simulator has to choose its own uops, which can be different from actual Westmere uops.

Moreover, memory subsystem error can also be reflected in execution time. Memory subsystem error has its roots in the misapproximation of cache delays and interconnect delays. Deviation from correct timing affects execution time. For example, if a cache line is not delivered on time, the pipeline stalls and the dependent instructions have to wait more. In a vicious chain of events, this also might cause more cache misses and more stalls.

MARSSx86 shows a very unsteady error as we change the number of threads and benchmarks. In Figure 14, the percentage of absolute error between the actual machine and the MARSSx86 total execution cycles for all three benchmarks is illustrated. Since we did not change the source code of benchmarks and since ptlcalls (functions to communicate between simulator and host machine) are not added to the source code, the core with the longest execution determines the execution time of the benchmark. The thread affinity is guaranteed since the number of available cores is set as the number of

threads. The error is variable from 9.82 percent for the 4-thread execution of Graph500 to 134.66 percent for the 4-thread execution of HeatPlate.

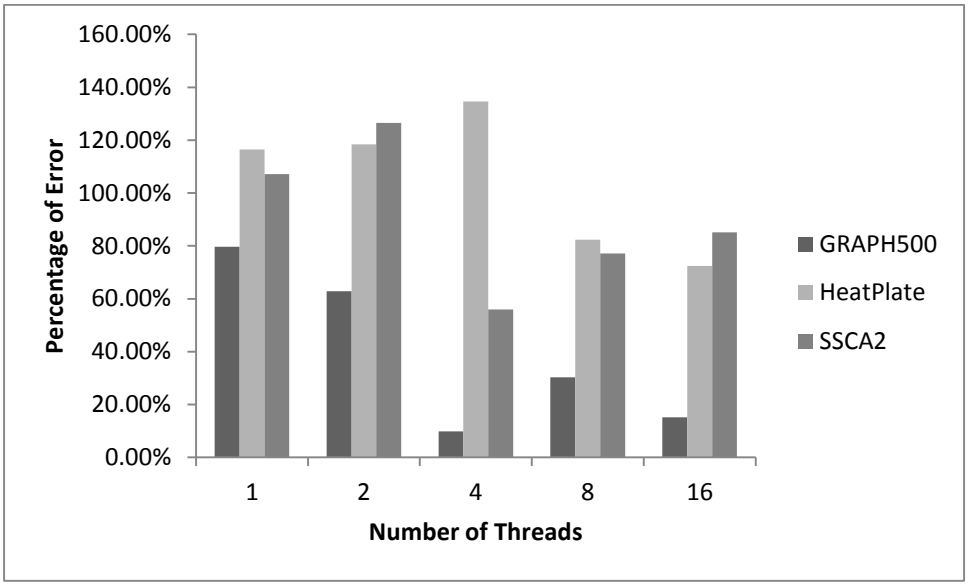

Figure 14- Error in total execution cycles

Although different error contributors in the pipeline are not decoupled and characterized, the amounts of miss rate error in L1, L2 and L3 are shown in Table 4, Table 5 and Table 6. Errors in this metric are a good representation of overall error in the memory subsystem model. The simulator memory model shows a variable error among executions with different number of threads and benchmarks. Since each execution of a benchmark takes days, multiple executions of benchmarks on the simulator was not possible. However, comparing the total execution error and cache errors definitively shows that the major source of error is not the cache hierarchy, but rather the pipeline.

Since the errors from other sources in pipeline are not needed to be characterized, more investigation on other metrics is not done.

	1	2	4	8	16
L1	25.37%	16.74%	14.85%	9.60%	35.27%
L2	14.37%	11.69%	20.03%	12.81%	202.08%
L3	96.42%	17.79%	41.96%	26.25%	240.17%

Table 4- Error of Cache Miss Rate for Graph500

	1	2	4	8	16
L1	1.07%	0.55%	19.30%	1.12%	19.35%
L2	1153.65%	1565.24%	946.71%	205.28%	68.22%
L3	2501.13%	2030.20%	99.60%	99.82%	99.43%

Table 5- Error of Cache Miss Rate for HeatPlate

	1	2	4	8	16
L1	12.06%	57.55%	65.32%	98.15%	143.36%
L2	2069.18%	99.93%	98.91%	98.45%	96.40%
L3	7976.56%	5729.60%	99.46%	99.32%	138.75%

Table 6- Error of Cache Miss Rate for SSCA2

The simulator shows an unacceptable error in 8 cases, which are marked. If we do not take these outliers into account, Figure 15 shows the average error of the cache miss rate in each cache level for each benchmark. The executions of HeatPlate and single

threaded Graph500 show a very large error in L2 cache. We do not have any specific explanation for this large error in the L2 cache model.

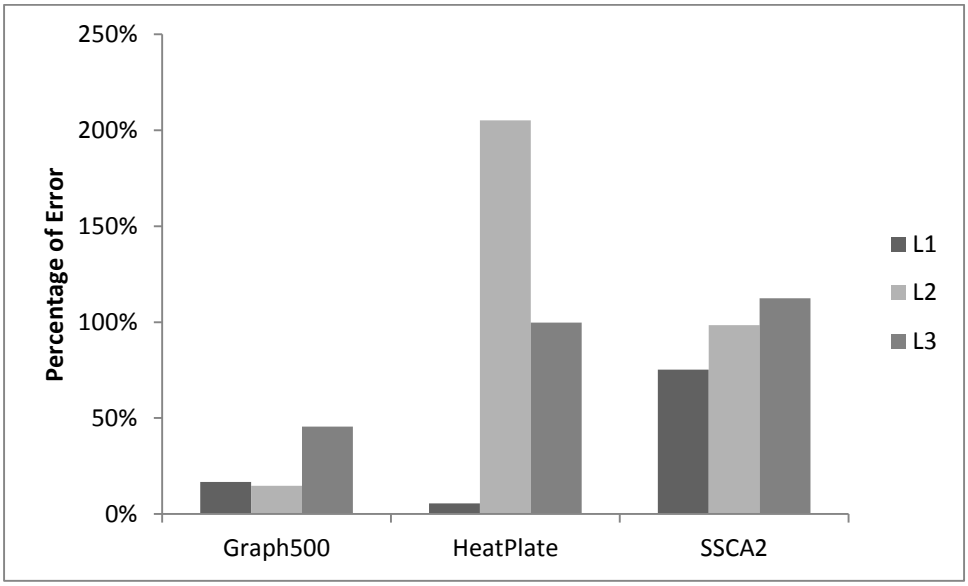

Figure 15- Average percent of error for each cache level

A large variable error in total execution cycles may cause error in overhead calculations. For example, a single threaded Graph500, which is used as the baseline performance of Graph500 (speedup=1) has an 80 percent error. This huge error in the baseline can change the result whenever we are measuring the speedup. Also, we have to measure the parallel and sequential portion of execution time from PGOMP in an actual machine because we are not able to measure them in simulation. Consequently, any deviation from actual machine timing may introduce an indistinguishable error to the overhead calculation.

5.3 INSTRUMENTATION TOOL RESULTS

As we mentioned in 4.1, lock and barrier time can be profiled as the application runs. An OpenMP instrumentation tool, PGOMP, is used to measure the barrier and lock time in runtime. Through ten executions, the speedup is measured. The ideal speedup is measured from Amdahl's law, while the sequential and parallel execution times are measured from PGOMP output. According to 2.3, overhead is calculated as the performance loss. In other words, overhead is the time gap between ideal and measured speedup. The lock overhead is the portion of this runtime overhead in which the application is waiting for critical section or lock contention (omp_set_lock function). Correspondingly, the barrier overhead is the percentage of performance loss in which threads wait on barriers.

Table 7 shows all of the resultant data from PGOMP on the actual machine. A scale of 16 is used for Graph500, a scale of 9 for SSCA2, and 300x300 points for Heatplate. The residual overhead is solely the coherency overhead, if we assume there is no other factor such as sequential variation overhead. Instead of calculating the sequential variation overhead as in [14], we used the samples with the highest lock and barrier overhead, which means the residual overhead is at a minimum. Moreover, Elfituri and Cook illustrated in [14] that parallelization management code in the OpenMP framework implementation does not contribute to performance loss. Consequently, this residual overhead has to be coherency protocol overhead since nothing else is left.

Application	Number of Threads	Measured Speedup	Ideal Speedup	Barrier Overhead %	Lock Overhead %	Residual Overhead %
Graph500	1	1	1	-	-	-
Graph500	2	1.695	1.946	35.81	0.00	64.19
Graph500	4	2.163	3.798	32.02	0.00	67.98
Graph500	8	3.709	7.327	42.86	0.00	57.14
Graph500	16	8.654	13.25	40.10	0.00	59.90
SSCA2	1	1	1	-	-	-
SSCA2	2	1.037	1.806	4.39	18.03	77.58
SSCA2	4	0.390	2.580	15.85	3.65	80.51
SSCA2	8	0.292	6.347	18.00	4.96	77.04
SSCA2	16	0.186	11.27	20.73	5.63	73.64
HeatPlate	1	1	1	-	-	-
HeatPlate	2	1.939	1.956	92.23	0.65	7.12
HeatPlate	4	3.489	3.557	85.78	1.86	12.36
HeatPlate	8	5.800	7.467	36.31	12.22	51.47
HeatPlate	16	6.440	13.80	34.93	5.18	59.89

Table 7- Contribution of barrier and lock to performance loss in actual machine

A comparison between barrier and lock overhead demonstrates that barrier contribution to performance loss is larger than lock in these scales of benchmarks. Since the simulator is not feasibly able to execute the benchmark in the available time, we have

to choose small scale benchmarks. This small dataset reduces the resource requirement enough to allow parallelization, and increases the contribution of barrier overhead. Also, as the execution time gets smaller, instrumentation noise has a bigger effect on the profiling data. This may explain lock overhead in the 8-threaded execution of HeatPlate, which disturbed its monotonic trend.

The last column shows an interesting behavior: Graph500 and SSCA2 both show a high and almost constant amount of cache coherency overhead, while HeatPlate has an increasing trend as we increase the number of threads. Finally, we can conclude that cache coherency is the largest contributor to the performance loss in graph applications.

5.4 DIRECT MEASUREMENT OF COHERENCY PROTOCOL OVERHEAD

MARSSx86, used as an x86 cycle-accurate full system simulator, enables us to directly measure the number of cycles that the cache has to spend to keep the cache data coherent. As we explained in 2.4, two cases should be measured:

1. *Read an invalidated data*. In this case, the requested cache line is present in the local cache, but it is invalidated by another processor. If the coherency issue did not exist, this memory reference request would have been a hit. However, since another processor invalidated this copy, it is a miss.
2. *Read miss to shared data*. In a single-core scenario, a miss in private cache is handled by sending a request to a lower level of memory (toward main memory).

However, in an SMP scenario, the valid copy of this memory reference might be either in lower memory or the other caches.

As soon as the memory request is made by any core, the request has been tracked, and if it is one of those two cases, the number of cycles from request creation to data delivery is recorded. Figure 16, Figure 17, and Figure 18 show the result of this MESI overhead measurement (Total execution cycles as well as MESI overhead as a percentage of total execution cycles are shown in Appendix C). Graph500 has a high and almost constant overall MESI overhead, while HeatPlate and SSCA2 have a sudden burst on 4 threads and 8 threads. More importantly, in all benchmarks, the overhead is dominated by *invalidated shared data* case. *Read miss to shared data* case has a monotonic increasing trend in all of the benchmarks as we increase the number of threads. Also, in SSCA2 and HeatPlate, an overall increasing trend can be seen in *invalidated shared data* case, although an irregular sudden increase can be seen at some points, such as 4-threaded execution of HeatPlate.

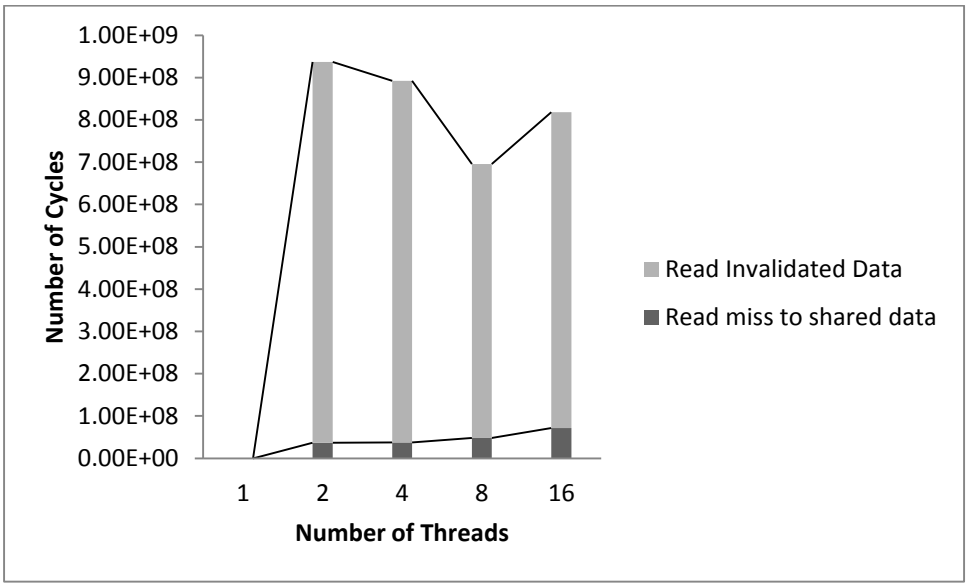

Figure 16- Graph500 MESI overhead

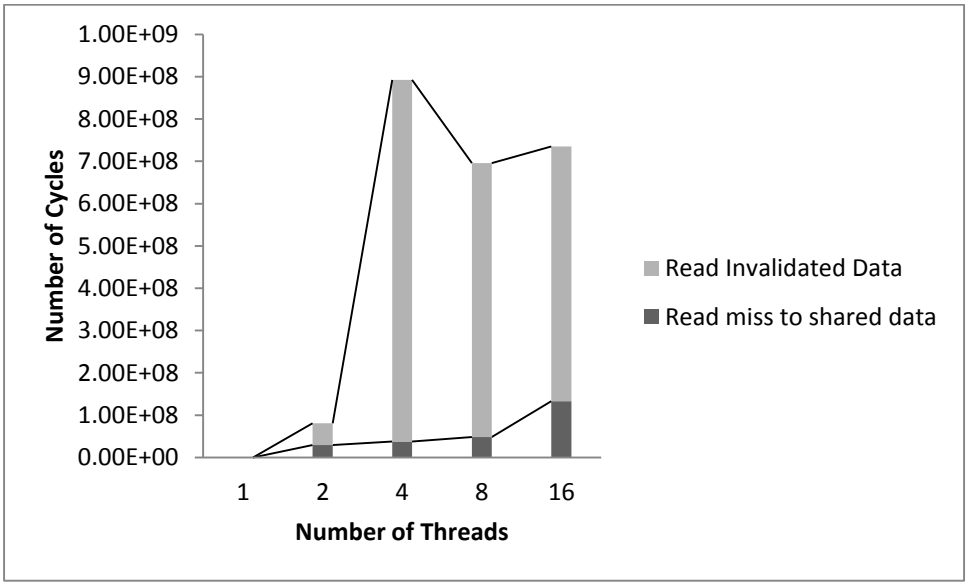

Figure 17- HeatPlate MESI overhead

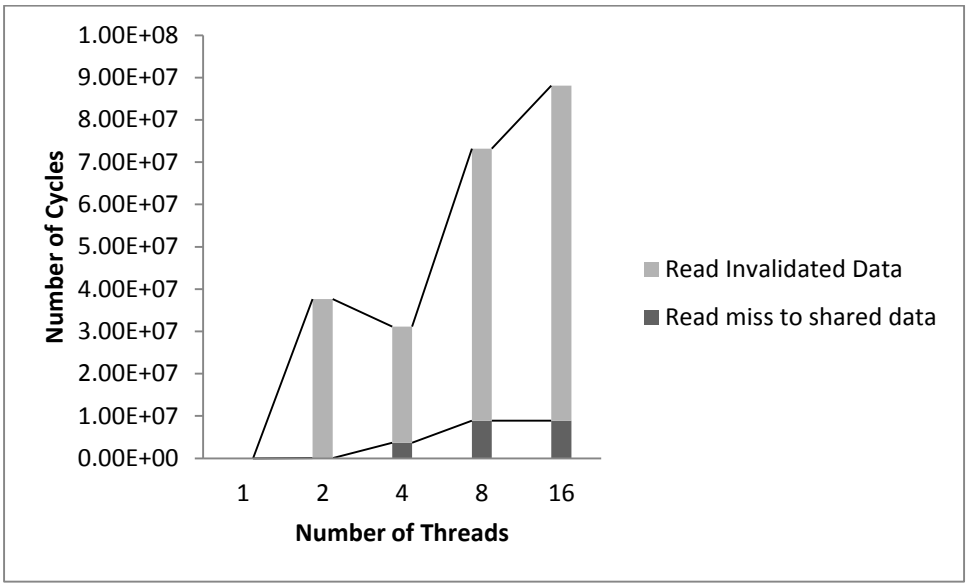

Figure 18- SSCA2 MESI overhead

5.5 ANALYSIS OF INTERCONNECT

Since graph applications such as Graph500 and SSCA2 are expected to have a large amount of un-core memory transfer, a drop in performance is expected as we increase the number of threads and utilize more cores. Although MARSSx86 is not designed to be a good interconnect model, we can still investigate some interesting facts in its results.

First, we should notice that Westmere-EX has a data path to L1 cache that is 16 bytes wide in each direction. Moreover, there is a 256 bit internal data path between L1 and L2, and the critical word is transferred first when a cache line is transferred. Consequently, the 8 byte data transfer delay is not different from the delay of 4-bytes

transfer. Since MARSSx86 simulates x86 architecture, 4 bytes are transferred and then the unused part is masked, regardless of the required data size.

However, transfer delay is expected to increase as the number of cores increases, because more cores try to access the interconnect media to send or receive data. It is expected that this problem will be exacerbated in graph applications, since they are data-intensive applications with high amounts of data read, write and un-core data transfers.

In the case that a core tries to read invalidated data and has to wait for another core to send a copy, we expect an increase in delay as the number of cores increases. Figure 19 and Figure 20 illustrate how the delay grows as the number of cores increases, and how the size of data is irrelevant since the data path is as big as 16 bytes. Figure 21, on the other hand, shows delay of read invalidated shared in HeatPlate. Interestingly, this plot does not show a high increasing slope as the number of threads increases. We see a small growth when the number of threads is increased from 8 to 16, but since the number of required transfers over time is not as large as a data-intensive application like Graph500, we cannot see a large increasing slope in its plot.

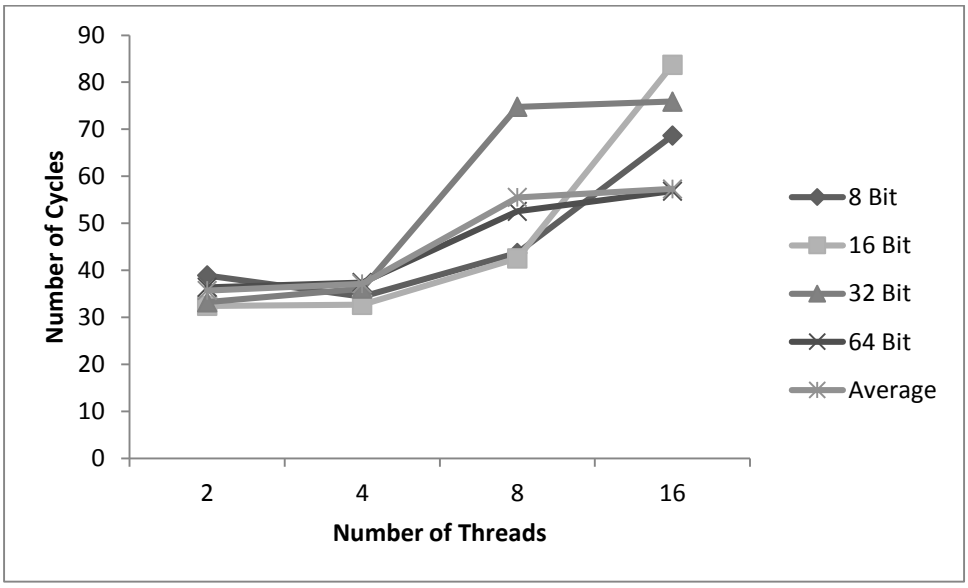

Figure 19- Delay of read invalidated shared in Graph500

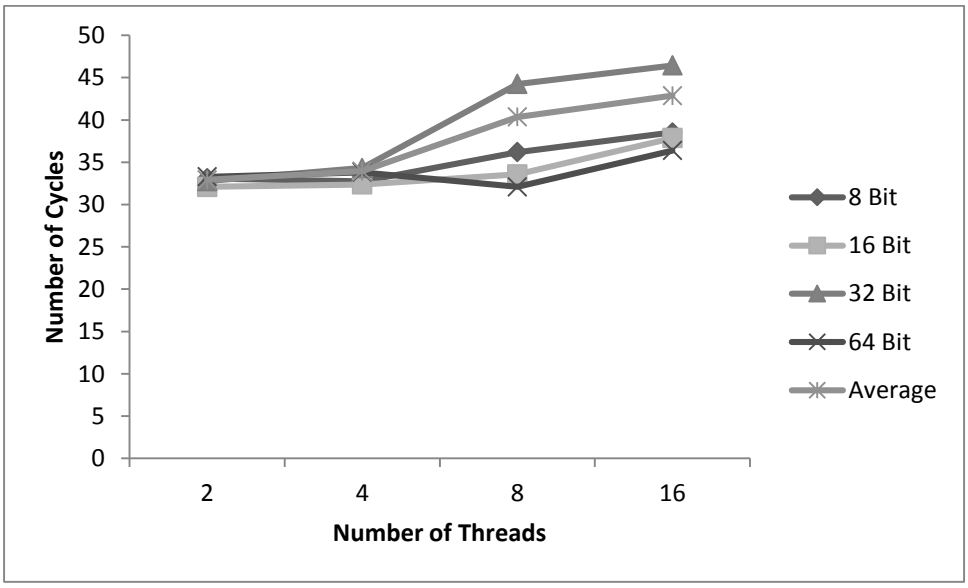

Figure 20- Delay of read invalidated shared in SSCA2

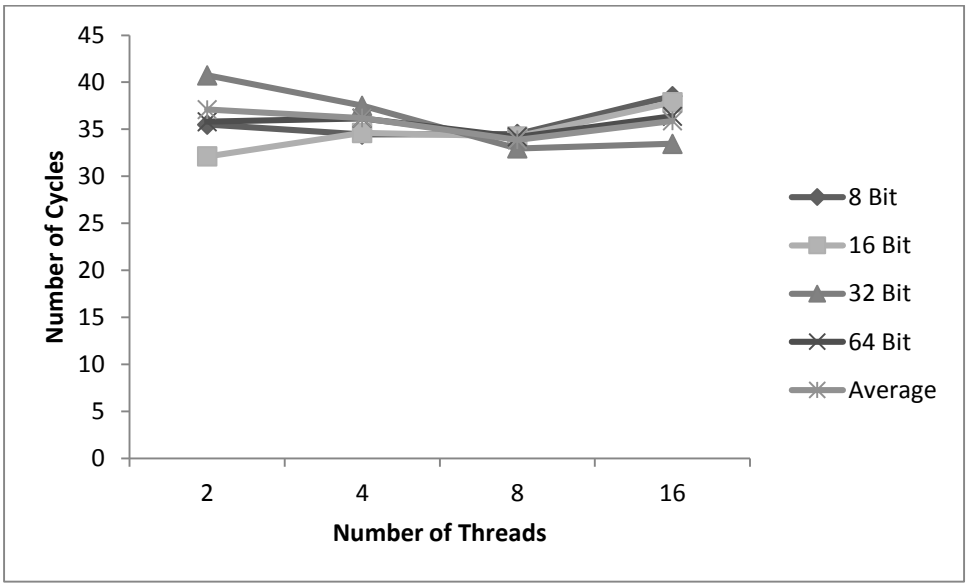

Figure 21- Delay of read invalidated shared in HeatPlate

Levinthal [64] did a performance evaluation in Xeon 5500 series, which has up to 4 cores. He approximated the difference between L3 hit delay in the case where data is shared and the case where the data is not shared by 25 cycles (~40 cycles if the cache line is not shared and ~65 if the cache line is shared). This approximation is close to our result from the MARSSx86 simulator, as the difference between the shared and unshared cache line case is an average of 15 cycles and a maximum of 35 cycles.

5.5.1 WRITE TO SHARED DATA

The third case of overhead, *write to a shared data*, is discussed in 4.4. In a split-phase bus media, data and snoopy traffic do not affect each other's bandwidth. Since no data transfer is needed in this case, it adds traffic only to snoopy bus. In a highly parallelized application where the data is shared among many threads, a large number of

store instructions may increase the traffic on snoopy bus and may challenge the interconnect performance. We are able to measure the interconnect performance when the snoopy messages increase. Nevertheless, we should consider the possibility that the interconnect model is not accurate, and may be slightly different from the Westmere interconnect.

After the invalidation request for a specific cache line is made, the media is acquired and the message is sent to the other cores. Any receiver cache puts the request into its queue and sends acknowledgement instantly. However, it takes time to invalidate that specific cache line based on queue traffic and cache speed. We use the term *invalidation delay* to refer to the gap in time between the creation of the invalidation request and the time when the cache line state is actually changed to invalid. This invalidation delay is measured for both the L1 and L2 caches. The invalidation delay depends on the interconnect topology and performance, the length of the cache snoopy message queue, and the cache delay. Figure 22 and Figure 23 show the invalidation delay in L1 and L2 as we increase the number of threads.

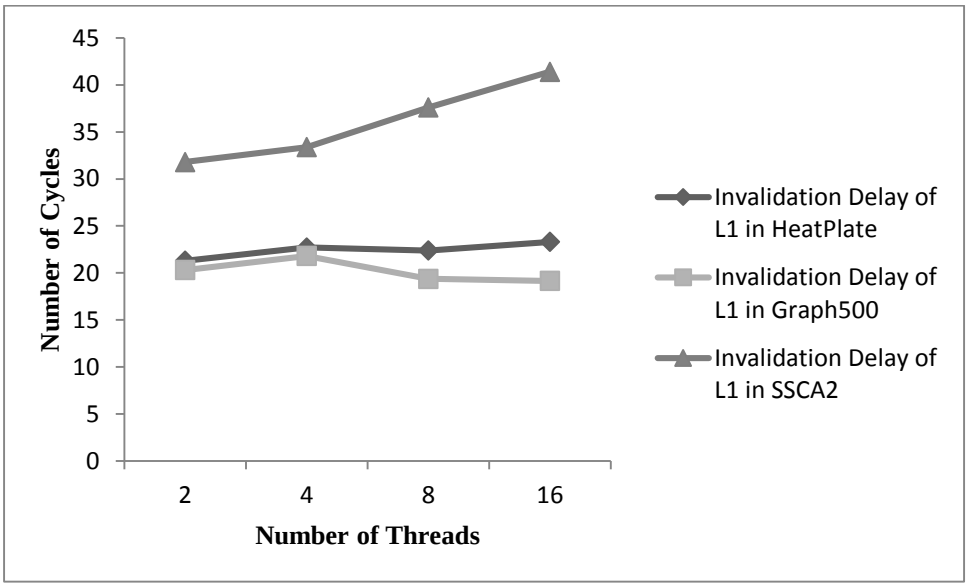

Figure 22- L1 invalidation delay in write to shared data

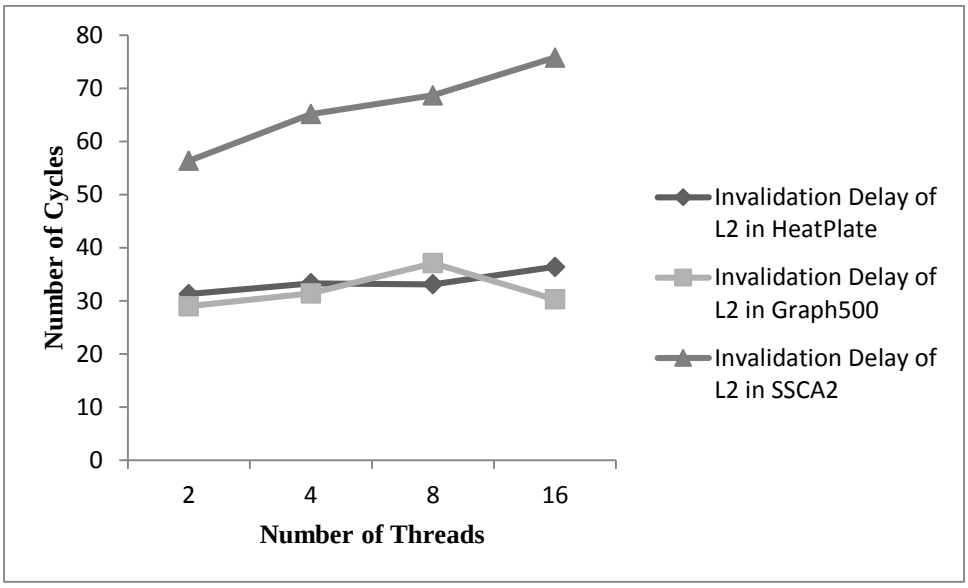

Figure 23- L2 invalidation delay in write to shared data

If we look at the SSCA2 curve, we see that both of these plots show a larger delay than Graph500 and HeatPlate. SSCA2 invalidation delay is also monotonically increasing

while the others are almost constant. This can be explained by the benchmark description that we provided in 4.5. Graph500 does not include intensive store instructions because it is a graph search, but SSCA2 contains many computation kernels, meaning that many store instructions are involved in SSCA2 code. Moreover, HeatPlate is not a data-intensive application, so a large amount of invalidation requests is not expected from it. When CPU cores execute SSCA2, they issue a large amount of invalidation requests to interconnect, causing the traffic to increase and exacerbating the delay. This can also explain the sensitivity of the SSCA2 curve, which is due to the number of threads. As the number of cores increases, more traffic is forced to interconnect, and this reduces the interconnect performance.

5.6 PERFORMANCE RESULTS FROM SIMULATION

The goal of this paper is to directly measure what percent of overall overhead is solely due to MESI protocol. Taking the execution time and the number of cycles which are spent to keep the cache coherent from simulator, as well as the ideal speedup for each number of threads from PGOMP output into account, and doing the same calculation as 4.1 and 5.3 provides the results in Figure 24, Figure 25 and Figure 26. This result almost validates the data from PGOMP. Considering the slightly high error in the total execution cycles, results from PGOMP and MARSSx86 are close enough in most of the cases. The minimum difference is in double-threaded execution of Graph500, where the difference between calculated and simulated MESI overhead is only 0.06 percent. The

maximum difference is in 16-threaded execution of SSCA, where the difference is 65.3 percent. However, simulation results do not validate PGOMP results in three cases: 16-threaded HeatPlate, 16-threaded SSCA2, and 16-threaded Graph500. SSCA2 is executed with a scale of 9, which is very small for this benchmark. It is possible that instrumentation noise changed the behavior and timing of the benchmark. At this small scale and with such a large number of threads, barrier and lock overhead are more expected. This means more investigations on PGOMP output for SSCA2 should be done.

Another reason that may cause this deviation in the 16-threaded execution of benchmarks is error in the baseline of total execution cycles. As we increase the number of threads and reduce the execution time, error can significantly change the amount of overall overhead. Obviously, since the numbers are smaller in this number of threads, the effect of error is even larger. Consequently, variable error may increase the overhead and reduce the percentage. Then, although the number of wasted cycles increases, the error leads to a smaller overhead percentage.

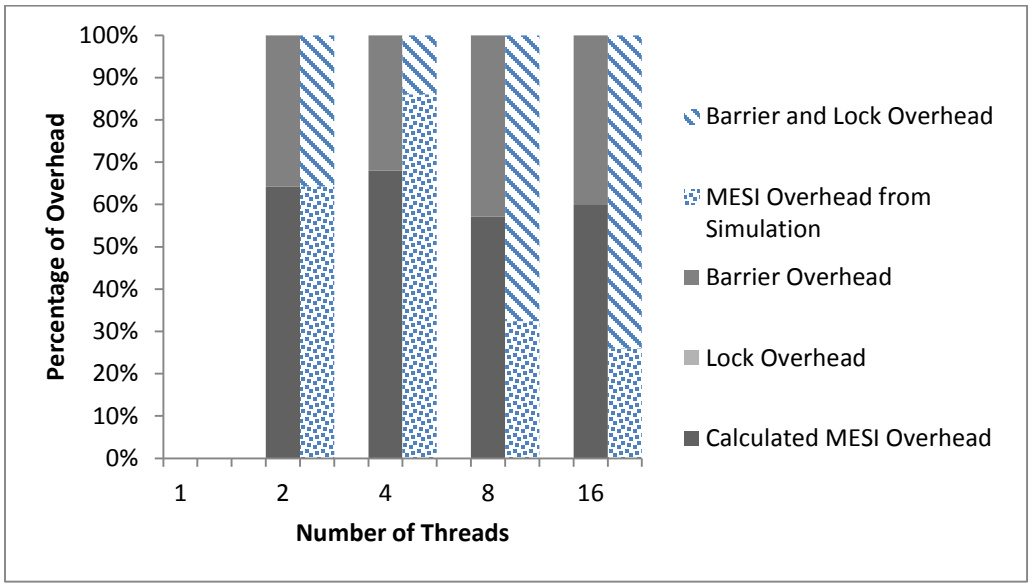

Figure 24- Measurement of MESI overhead in Graph500 from simulator

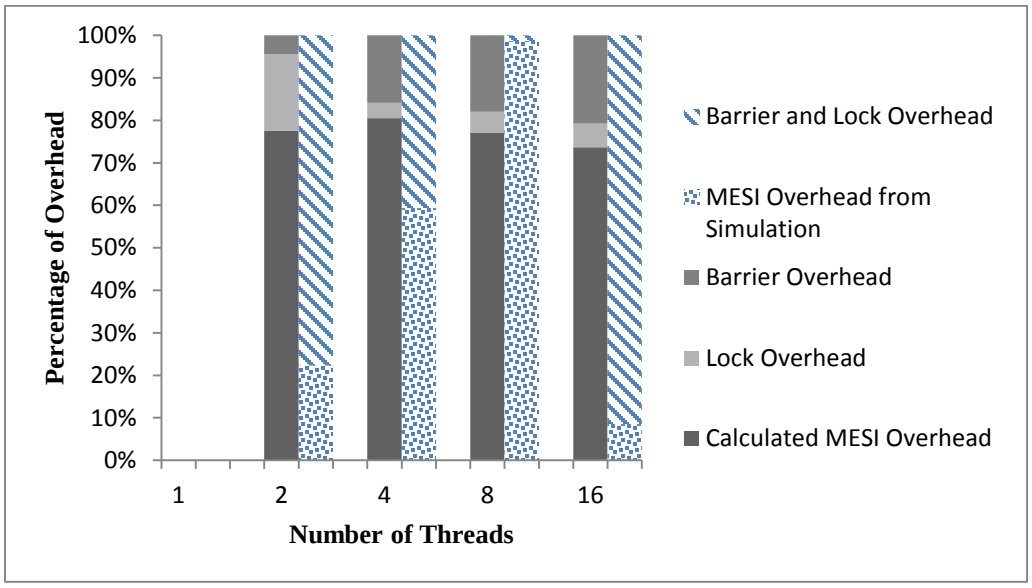

Figure 25- Measurement of MESI overhead in SSCA2 from simulator

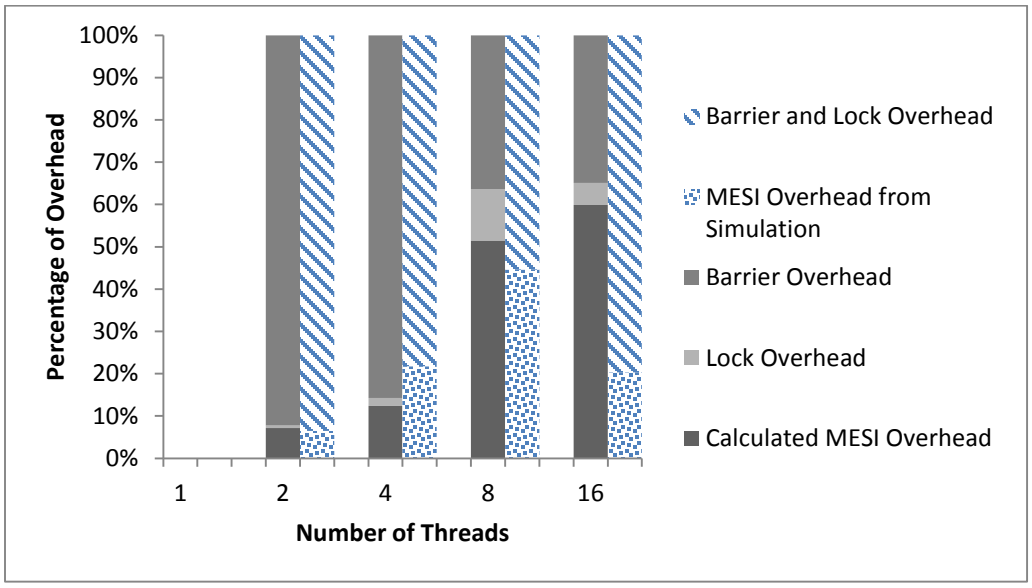

Figure 26- Measurement of MESI overhead in HeatPlate from simulator

Besides 16-threaded execution of benchmarks, which has been discussed, the rest of the cases validate the trend that results from the actual machine, although, compared to instrumentation results, Graph500 has up to a 25 percent difference, HeatPlate up to a 20 percent difference, and SSCA2 has up to a 10 percent difference.

Table 8 shows the concluding results from putting together all the measurements in simulator. The measured speedup column is calculated by total execution cycles from simulator. *MESI overhead percent column* is the percent of contribution to performance loss from the memory coherency protocol. The residual unobserved overhead must due to lock and synchronization. We can conclude that the simulation confirms that MESI protocol overhead is a major reason for performance loss in graph applications.

Application	Number of Threads	Measured Speedup	Ideal Speedup	MESI Overhead %	Residual Overhead %
Graph500	1	1	1	-	-
Graph500	2	1.784	1.946	35.73	64.27
Graph500	4	3.376	3.798	13.98	86.02
Graph500	8	4.877	7.327	67.56	32.44
Graph500	16	7.027	13.25	73.99	26.01
SSCA2	1	1	1	-	-
SSCA2	2	0.948	1.806	22.32	77.68
SSCA2	4	1.840	2.580	59.32	40.68
SSCA2	8	2.647	6.347	98.69	1.31
SSCA2	16	0.317	11.27	8.54	91.46
HeatPlate	1	1	1	-	-
HeatPlate	2	1.922	1.956	6.50	93.50
HeatPlate	4	3.219	3.557	21.76	78.24
HeatPlate	8	6.884	7.467	44.61	55.39
HeatPlate	16	8.086	13.80	20.34	79.66

Table 8- Contribution of MESI overhead to performance loss in simulator

5.7 FUTURE WORK

To measure the overhead more accurately, several improvements can be done:

1. Besides using a more accurate simulator, more accurate benchmarking can improve the reliability of results. If ptlcals were added inside the benchmark code directly, we would be able to measure execution time and overhead specifically for desired parts of code. This would allow a better approximation of parallel and sequential parts, ideal speedup, and also measured speedup. This could significantly improve accuracy.
2. The biggest constraint for simulations is time. Since it takes weeks to simulate a benchmark at cycle-accuracy, we had to use small scale benchmarks. If we reduce the accuracy by using only a cycle-accurate memory subsystem simulator instead of a full system simulator, we will be able to simulate larger scales. Larger scale benchmarks show more stable results and speedup. Also, as the execution times grow, the effect of bash noise and instrumentation noise decreases, and profiling software results are more reliable.
3. Using a *core Performance Monitoring Unit (PMU)* is another option. Many PMU counters are not accurate enough to be used for quantitative calculation, especially when we are looking for memory subsystem counters. Also, they may not directly measure what we need. However, they may be more accurate than simulators, and conclusions from different counters may help us to measure what we need.

4. The accuracy of PGOMP should be investigated. We do not know how accurate PGOMP is and how much instrumentation error is forced to runtime execution. Also, PGOMP is supposed to measure only the OpenMP function's execution time, but adding more time profiling, such as overall execution time, can improve its accuracy. Also, profiling in number of cycles instead of time can improve accuracy tremendously, since we can use PAPI output as well as PGOMP output.

5.8 SUMMARY

This chapter begins with an evaluation of a spatial locality metric for three benchmarks: Graph500, SSCA, and HeatPlate. This metric showed that Graph500 and SSCA, as graph benchmarks, have low levels of locality in comparison with the other multithreaded applications. Simulator accuracy is discussed, along with its effect on final results. Coherency protocol overhead is calculated by an instrumentation tool and validated by direct measurement from the simulator.

6 CONCLUSION

Memory performance has not been improved by the same speed as processor performance. This fact leads us to a tendency toward integrating caches – small, high speed memories - into processors. We also prefer to organize algorithms and hardware to take advantage of spatial and temporal locality to increase the performance of these caches. Moreover, in a scenario where we have multiple processing units, we have an additional complexity. Since some part of data has to be shared among these processing units, they should follow a protocol to keep the data coherent. MESI is the most common cache coherency protocol in multicore systems.

Graph benchmarks are known as applications with a low level of locality. Graph500 and SSCA2 are graph benchmarks that are used in this paper. HeatPlate is another multithreaded benchmark, which is used to represent regular parallel applications. A simple normalized spatial locality metric is used to show how they have different levels of locality. Graph500 and SSCA2 both showed very low spatial locality, while HeatPlate showed a moderate spatial locality.

According to Amdahl’s law, we are able to increase the speed-up associated with the parallel portion of application. However, measurements by instrumentation tools showed that, even if we omit the sequential part, the ideal speedup is still unreachable. This gap between ideal and measured speedup is named parallelization overhead. A

discussion about the source of this overhead has been made. Then, PGOMP as an instrumentation tool is used to profile the OpenMP benchmarks.

Data from the simulator, as well as profiling data from the instrumentation tool in actual machine cleared that barrier, critical section, and lock contention are the source of 20 to 93 percent of performance loss in our benchmarks. Considering possible reasons that may cause such a performance loss, we can conclude that the residual percentage is due to coherency protocol overhead.

Since cycle-accurate information from coherency protocols is unreachable in actual machines, we used a cycle accurate simulator, MARSSx86, to directly measure this coherency protocol overhead. The simulator showed a variable error in execution of the three benchmarks. Simulator error in total execution cycles varied from 9 to 134 percent. This error can be root of further errors in overhead calculations.

Three cases are considered as MESI protocol overhead: *read invalidate shared data*, *read miss to shared data*, and *write to shared data*. The last one cannot affect data bandwidth. However, it is shown that a high amount of snoopy message traffic can increase the invalidation delay up to 12 cycles in the SSCA2 benchmark, which has a large amount of store operations. We also showed that MESI protocol overhead is dominated by the read invalidated data case.

Finally, the same calculation as PGOMP data has been done on simulation results. MESI overhead from simulation results are from 0.05 percent to 65 percent different

from PGOMP output. However, they validate the trend of the other set of data. They also confirm that coherency protocol overhead is a large portion of overhead in the graph benchmarks that we studied, since they are source of up to 90 percent of performance loss.

APPENDIX A: MARSSX86 MACHINE CONFIGURATION FILE

```
core:
  ooo:
    base: ooo
    params:
      ISSUE_WIDTH: 5
      COMMIT_WIDTH: 4
      ROB_SIZE: 128
      ISSUE_Q_SIZE: 36
      ALU_FU_COUNT: 6
      FPU_FU_COUNT: 6
      LOAD_FU_COUNT: 1
      STORE_FU_COUNT: 1
      LOAD_Q_SIZE: 48
      STORE_Q_SIZE: 32
cache:
  l1_32k_8_:
    base: mesi_cache
    params:
      SIZE: 32K
```

```
    LINE_SIZE: 64 # bytes
    ASSOC: 8
    LATENCY: 4
    READ_PORTS: 2
    WRITE_PORTS: 1
l1_32k_4_:
  base: mesi_cache
  params:
    SIZE: 32K
    LINE_SIZE: 64 # bytes
    ASSOC: 4
    LATENCY: 2
    READ_PORTS: 2
    WRITE_PORTS: 1
l2_256k:
  base: mesi_cache
  params:
    SIZE: 256K
    LINE_SIZE: 64 # bytes
    ASSOC: 8
```

```
      LATENCY: 6
      READ_PORTS: 2
      WRITE_PORTS: 2
    l3_24M:
      base: wb_cache
      params:
        SIZE: 24M
        LINE_SIZE: 64 # bytes
        ASSOC: 24
        LATENCY: 27
        READ_PORTS: 2
        WRITE_PORTS: 2

memory:
  dram_cont:
    base: simple_dram_cont
machine:
  xeon1:
    description: Mix of OOO and Atom cores with private L2
    min_contexts: 1
```

```
cores:
  - type: ooo
    name_prefix: ooo_
    option:
        threads: 1
caches:
  - type: l1_32k_4_
    name_prefix: L1_I_
    insts: $NUMCORES
    option:
        private: true
  - type: l1_32k_8_
    name_prefix: L1_D_
    insts: $NUMCORES
    option:
        private: true
  - type: l2_256k
    name_prefix: L2_
    insts: $NUMCORES
    option:
```

```
        private: true
        last_private: true
    - type: l3_24M
      name_prefix: L3_
      insts: 1
memory:
    - type: dram_cont
      name_prefix: MEM_
      insts: 1 # Single DRAM controller
      option:
        latency: 54 # In nano seconds
interconnects:
    - type: p2p
      connections:
        - core_$: I
          L1_I_$: UPPER
        - core_$: D
          L1_D_$: UPPER
        - L1_I_$: LOWER
          L2_$: UPPER
```

```
    - L1_D_$: LOWER
      L2_$: UPPER2
    - L3_0: LOWER
      MEM_0: UPPER
- type: split_bus
  connections:
    - L2_*: LOWER
      L3_0: UPPER
```

APPENDIX B: MARSSX86 CORE CONFIGURATION FILE

```
#ifndef OOOCORE_CONST_H

#define OOOCORE_CONST_H

#ifndef OOO_ISSUE_WIDTH

#define OOO_ISSUE_WIDTH 4

#endif

#ifndef OOO_MAX_PHYS_REG_FILE_SIZE

#define OOO_MAX_PHYS_REG_FILE_SIZE 256

#endif

#ifndef OOO_PHYS_REG_FILE_SIZE

#define OOO_PHYS_REG_FILE_SIZE 256

#endif

#ifndef OOO_BRANCH_IN_FLIGHT

#define OOO_BRANCH_IN_FLIGHT 24

#endif

#ifndef OOO_LOAD_Q_SIZE

#define OOO_LOAD_Q_SIZE 48

#endif

#ifndef OOO_STORE_Q_SIZE

#define OOO_STORE_Q_SIZE 48
```

```
#endif
#ifndef OOO_FETCH_Q_SIZE
#define OOO_FETCH_Q_SIZE 48
#endif
#ifndef OOO_ISSUE_Q_SIZE
#define OOO_ISSUE_Q_SIZE 64
#endif
#ifndef OOO_ROB_SIZE
#define OOO_ROB_SIZE 128
#endif
#ifndef OOO_FETCH_WIDTH
#define OOO_FETCH_WIDTH 4
#endif
#ifndef OOO_FRONTEND_WIDTH
#define OOO_FRONTEND_WIDTH 4
#endif
#ifndef OOO_FRONTEND_STAGES
#define OOO_FRONTEND_STAGES 4
#endif
#ifndef OOO_DISPATCH_WIDTH
```

```
#define OOO_DISPATCH_WIDTH 4
#endif
#ifndef OOO_WRITEBACK_WIDTH
#define OOO_WRITEBACK_WIDTH 4
#endif
#ifndef OOO_COMMIT_WIDTH
#define OOO_COMMIT_WIDTH 4
#endif
#ifndef OOO_ITLB_SIZE
#define OOO_ITLB_SIZE 32
#endif
#ifndef OOO_DTLB_SIZE
#define OOO_DTLB_SIZE 32
#endif
/* functional units */
#ifndef OOO_ALU_FU_COUNT
#define OOO_ALU_FU_COUNT 2
#endif
#ifndef OOO_FPU_FU_COUNT
#define OOO_FPU_FU_COUNT 2
```

```
#endif
#ifndef OOO_LOAD_FU_COUNT
#define OOO_LOAD_FU_COUNT 2
#endif
#ifndef OOO_STORE_FU_COUNT
#define OOO_STORE_FU_COUNT 2
#endif
#ifndef OOO_LOADLAT
#define OOO_LOADLAT 2
#endif
#ifndef OOO_ALULAT
#define OOO_ALULAT 1 /* ALU latency, assuming fast bypass */
#endif
/* max resources - Non configurable */
#define OOO_MAX_FU_COUNT 16
namespace OOO_CORE_MODEL {
   static const int MAX_THREADS_BIT = 4; /* up to 16 threads */
   static const int MAX_ROB_IDX_BIT = 12; /* up to 4096 ROB entries */
   /*
    * Operand formats
```

```
 */
static const int MAX_OPERANDS = 4;
static const int RA = 0;
static const int RB = 1;
static const int RC = 2;
static const int RS = 3; /* (for stores only) */
/*
 * Uop to functional unit mappings
 */
const int FU_COUNT = OOO_MAX_FU_COUNT;
const int ALU_FU_COUNT = OOO_ALU_FU_COUNT;
const int FPU_FU_COUNT = OOO_FPU_FU_COUNT;
const int STORE_FU_COUNT = OOO_STORE_FU_COUNT;
const int LOAD_FU_COUNT = OOO_LOAD_FU_COUNT;
const int LOADLAT = OOO_LOADLAT;
const int ALULAT = OOO_ALULAT;
/*
 * Global limits
 */
const int MAX_ISSUE_WIDTH = OOO_ISSUE_WIDTH;
```

```
        /* Largest size of any physical register file or the store queue: */
        const int MAX_PHYS_REG_FILE_SIZE =
OOO_MAX_PHYS_REG_FILE_SIZE;
        //  const int PHYS_REG_FILE_SIZE = 256;
        const int PHYS_REG_FILE_SIZE = OOO_PHYS_REG_FILE_SIZE;
        const int PHYS_REG_NULL = 0;
        enum { PHYSREG_NONE, PHYSREG_FREE, PHYSREG_WAITING,
PHYSREG_BYPASS,
           PHYSREG_WRITTEN, PHYSREG_ARCH, PHYSREG_PENDINGFREE,
MAX_PHYSREG_STATE };
        /*
         *
         * IMPORTANT! If you change this to be greater than 256, you MUST
         * #define BIG_ROB below to use the correct associative search logic
         * (16-bit tags vs 8-bit tags).
         *
         * SMT always has BIG_ROB enabled: high 4 bits are used for thread id
         */
      #define BIG_ROB
        const int ROB_SIZE = OOO_ROB_SIZE;
```

```
    /* const int ROB_SIZE = 64; */
    /* Maximum number of branches in the pipeline at any given time */
    const int MAX_BRANCHES_IN_FLIGHT = OOO_BRANCH_IN_FLIGHT;
    /* Set this to combine the integer and FP phys reg files: */
    /* #define UNIFIED_INT_FP_PHYS_REG_FILE */
#ifdef UNIFIED_INT_FP_PHYS_REG_FILE
    /* unified, br, st */
    const int PHYS_REG_FILE_COUNT = 3;
#else
    /* int, fp, br, st */
    const int PHYS_REG_FILE_COUNT = 4;
#endif
    /*
     * Load and Store Queues
     */
    const int LDQ_SIZE = OOO_LOAD_Q_SIZE;
    const int STQ_SIZE = OOO_STORE_Q_SIZE;
    /*
     * Fetch
     */
```

```
const int FETCH_QUEUE_SIZE = OOO_FETCH_Q_SIZE;
const int FETCH_WIDTH = OOO_FETCH_WIDTH;
/*
 * Frontend (Rename and Decode)
 */
const int FRONTEND_WIDTH = OOO_FRONTEND_WIDTH;
const int FRONTEND_STAGES = OOO_FRONTEND_STAGES;
/*
 * Dispatch
 */
const int DISPATCH_WIDTH = OOO_DISPATCH_WIDTH;
/*
 * Writeback
 */
const int WRITEBACK_WIDTH = OOO_WRITEBACK_WIDTH;
/*
 * Commit
*/
const int COMMIT_WIDTH = OOO_COMMIT_WIDTH;
      // #define MULTI_IQ
```

```
        // #ifdef ENABLE_SMT
      /*
       * Multiple issue queues are currently only supported in
       * the non-SMT configuration, due to ambiguities in the
       * ICOUNT SMT heuristic when multiple queues are active.
       */
        // #undef MULTI_IQ
        // #endif
#ifdef MULTI_IQ
    const int MAX_CLUSTERS = 4;
    /*
     * Clustering, Issue Queues and Bypass Network
     */
    const int MAX_FORWARDING_LATENCY = 2;
    static const int ISSUE_QUEUE_SIZE = 16;
#else
    const int MAX_CLUSTERS = 1;
    const int MAX_FORWARDING_LATENCY = 0;
    static const int ISSUE_QUEUE_SIZE = OOO_ISSUE_Q_SIZE;
#endif
```

```
        /* TLBs */
        const int ITLB_SIZE = OOO_ITLB_SIZE;
        const int DTLB_SIZE = OOO_DTLB_SIZE;
        /* How many bytes of x86 code to fetch into decode buffer at once */
        static const int ICACHE_FETCH_GRANULARITY = 16;
        /* Deadlock timeout: if nothing dispatches for this many cycles, flush the
pipeline */
        static const int DISPATCH_DEADLOCK_COUNTDOWN_CYCLES = 4096;
//256;
        /* Size of unaligned predictor Bloom filter */
        static const int UNALIGNED_PREDICTOR_SIZE = 4096;
        /* String names used in stats labels */
        extern const char* physreg_state_names[MAX_PHYSREG_STATE];
        extern const char* short_physreg_state_names[MAX_PHYSREG_STATE];
      #ifdef MULTI_IQ
        extern const char* cluster_names[MAX_CLUSTERS];
      #else
        extern const char* cluster_names[MAX_CLUSTERS];
      #endif
        extern const char* phys_reg_file_names[PHYS_REG_FILE_COUNT];
```

```
        };

#endif /* OOOCORE_CONST_H */
```

APPENDIX C: TOTAL EXECUTION CYCLES IN SIMULATOR

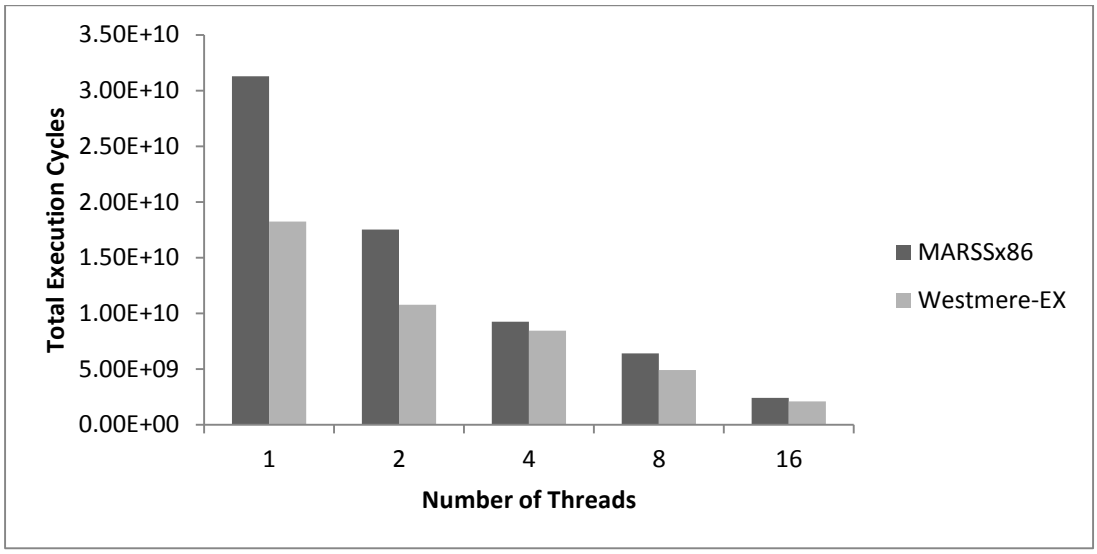

Figure 27- Graph500 total execution cycles in MARSSx86

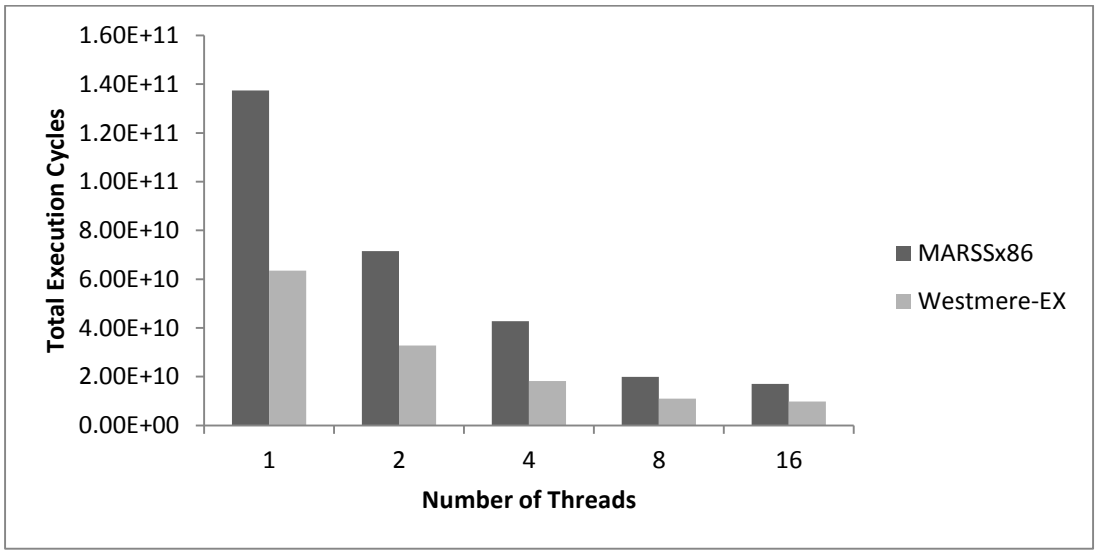

Figure 28- HeatPlate total execution cycles in MARSSx86

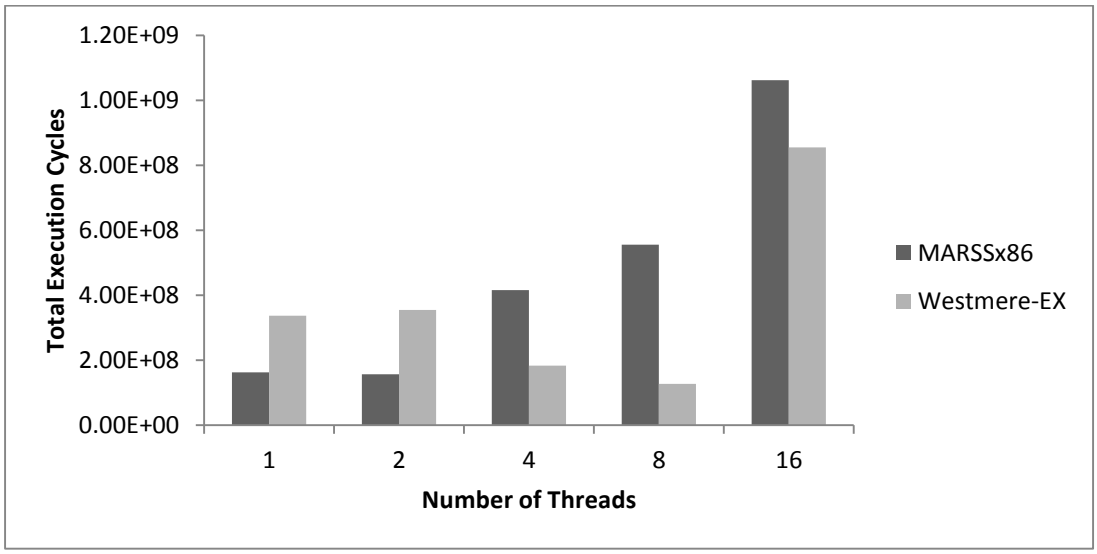

Figure 29- SSCA2 total execution cycles in MARSSx86

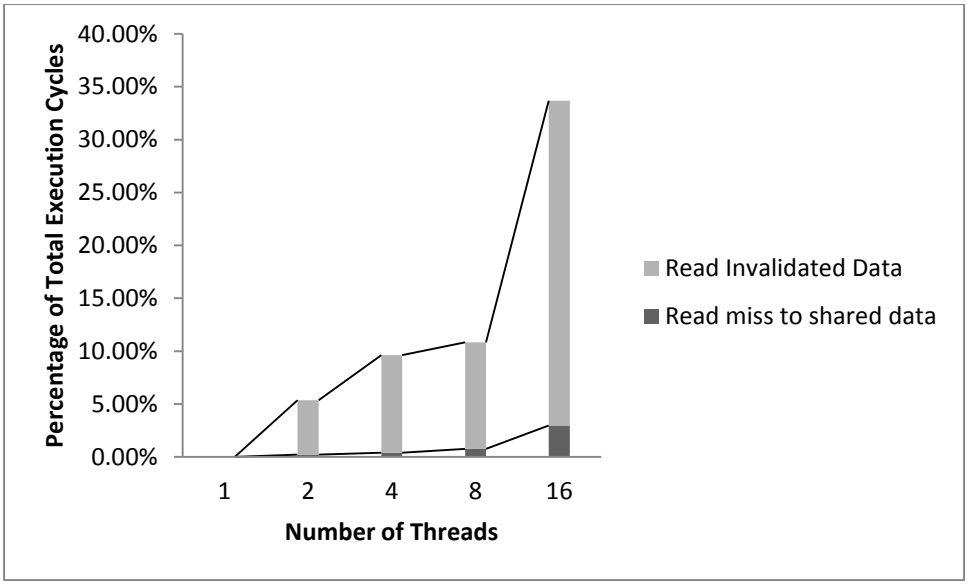

Figure 30- Graph500 MESI Overhead

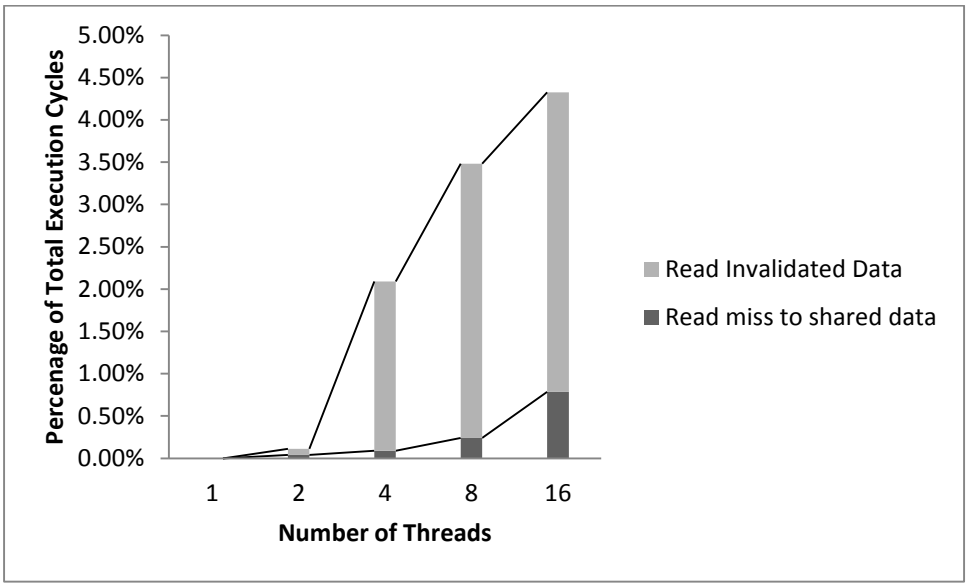

Figure 31- HeatPlate MESI Overhead

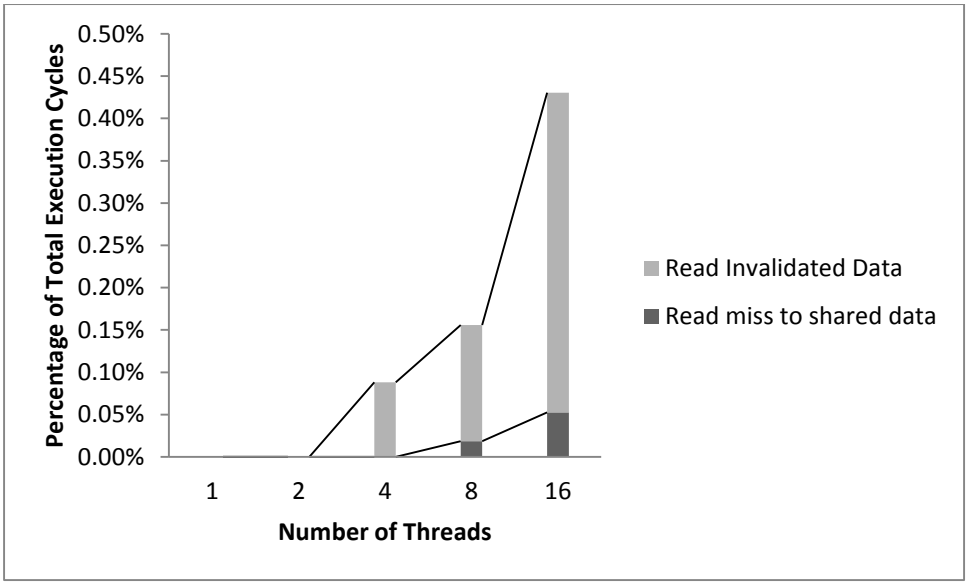

Figure 32- SSCA2 MESI overhead

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Morgan Kaufmann, 1996.

[2] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH Comput. Archit. news*, vol. 23, no. 1, pp. 20–24, 1995.

[3] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, *Clock rate versus IPC: The end of the road for conventional microarchitectures*, vol. 28, no. 2. ACM, 2000.

[4] M. R. Marty, *Cache coherence techniques for multicore processors*. ProQuest, 2008.

[5] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, and S. W. Williams, "The landscape of parallel computing research: A view from Berkeley," Technical Report *UCB/EECS-2006-183*, EECS Department, University of California, Berkeley, 2006.

[6] P. Stenstrom, "A survey of cache coherence schemes for multiprocessors," *Computer (Long. Beach. Calif).*, vol. 23, no. 6, pp. 12–24, 1990.

[7] R. Lawrence, "A Survey of Cache Coherence Mechanisms in Shared Memory Multiprocessors," *Dep. Comput. Sci. Univ. Manitoba, Manitoba, Canada*, 1998.

[8] D. J. Lilja, "Cache coherence in large-scale shared-memory multiprocessors:

Issues and comparisons," *ACM Comput. Surv.*, vol. 25, no. 3, pp. 303–338, 1993.

[9] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood, "Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, 2003, pp. 206–217.

[10] M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood, "Bandwidth adaptive snooping," in *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, 2002, pp. 251–262.

[11] J. R. Goodman, "Using cache memory to reduce processor-memory traffic," in *ACM SIGARCH Computer Architecture News*, 1983, vol. 11, no. 3, pp. 124–131.

[12] "An Introduction to the Intel® QuickPath", Intel, January 2009. [Online]. Available: http://www.intel.com/technology/quickpath/introduction.pdf

[13] D. Ziakas, A. Baum, R. A. Maddox, and R. J. Safranek, "Intel® QuickPath Interconnect Architectural Features Supporting Scalable System Architectures," *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*. pp. 1–6, 2010.

[14] M. Elfituri, J.Cook, and J. Cook, "Characterizing Performance Issues in Shared Memory Parallel Graph Benchmarks," unpublished.

[15] M. E. Crovella and T. J. LeBlanc, "Parallel performance prediction using lost cycles analysis," in *Proceedings of the 1994 ACM/IEEE conference on*

Supercomputing, 1994, pp. 600–609.

[16] T. H. Dunigan, “Kendall Square multiprocessor: Early experiences and performance,” *Citeseer*, 1992.

[17] M. Roth, M. J. Best, C. Mustard, and A. Fedorova, “Deconstructing the overhead in parallel applications,” in *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, 2012, pp. 59–68.

[18] R. C. Kunz, “Performance bottlenecks on large-scale shared-memory multiprocessors.” *Citeseer*, 2004.

[19] M. K. Prabhu and K. Olukotun, “Using thread-level speculation to simplify manual parallelization,” *ACM SIGPLAN Not.*, vol. 38, no. 10, pp. 1–12, 2003.

[20] S. V Adve and K. Gharachorloo, “Shared memory consistency models: A tutorial,” *Computer (Long. Beach. Calif).*, vol. 29, no. 12, pp. 66–76, 1996.

[21] T.-F. C. T.-F. Chen and J.-L. B. J.-L. Baer, “Effective hardware-based data prefetching for high-performance processors,” *IEEE Trans. Comput.*, vol. 44, 1995.

[22] J.-L. Baer and T.-F. C. T.-F. Chen, “An effective on-chip preloading scheme to reduce data access penalty,” *Proc. 1991 ACM/IEEE Conf. Supercomput. (Supercomputing ’91)*, 1991.

[23] B. Calder, C. Krintz, S. John, and T. Austin, “Cache-conscious data placement,” *ACM SIGOPS Operating Systems Review*, vol. 32. pp. 139–149, 1998.

[24] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2012.

[25] C. Pyo, K.-W. Lee, H.-K. Han, and G. Lee, "Reference distance as a metric for data locality," in *High Performance Computing on the Information Superhighway, 1997. HPC Asia'97*, 1997, pp. 151–156.

[26] K. Beyls and E. D'Hollander, "Reuse distance as a metric for cache behavior," in *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, 2001, vol. 14, pp. 350–360.

[27] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9. pp. 78–117, 1970.

[28] Q. Niu, J. Dinan, Q. Lu, and P. Sadayappan, "PARDA: A Fast Parallel Reuse Distance Analysis Algorithm," *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. pp. 1284–1294, 2012.

[29] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *ACM SIGPLAN Notices*, 2003, vol. 38, no. 5, pp. 245–257.

[30] M. Snir and J. Yu, "On the theory of spatial and temporal locality," 2005, [Online]. Available: https://www.ideals.illinois.edu/handle/2142/11077

[31] A. Anghel, G. Dittmann, R. Jongerius, and R. P. Luijten, "Spatio-Temporal Locality Characterization." [Online]. Available: http://www.cs.utah.edu/wondp/Anghel_Locality.pdf

[32] J. Weinberg, M. O. McCracken, E. Strohmaier, and A. Snavely, “Quantifying locality in the memory access patterns of hpc applications,” in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, 2005, p. 50.

[33] J. Hennessy, M. Heinrich, and A. Gupta, “Cache-coherent distributed shared memory: perspectives on its development and future challenges,” *Proc. IEEE*, vol. 87, no. 3, pp. 418–429, 1999.

[34] E. A. Emerson and E. M. Clarke, *Characterizing correctness properties of parallel programs using fixpoints*. Springer, 1980.

[35] M. Chaudhuri and M. Heinrich, “The impact of negative acknowledgments in shared memory scientific applications,” *Parallel Distrib. Syst. IEEE Trans.*, vol. 15, no. 2, pp. 134–150, 2004.

[36] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J. P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, and M. Horowitz, “The performance impact of flexibility in the Stanford FLASH multiprocessor,” in *ACM SIGPLAN Notices*, 1994, vol. 29, no. 11, pp. 274–285.

[37] M. Heinrich, V. Soundararajan, J. Hennessy, and A. Gupta, “A quantitative analysis of the performance and scalability of distributed shared memory cache coherence protocols,” *Comput. IEEE Trans.*, vol. 48, no. 2, pp. 205–217, 1999.

[38] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, “Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system,” in

Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on, 2009, pp. 261–270.

[39] L. Peng, J.-K. Peir, T. K. Prakash, C. Staelin, Y.-K. Chen, and D. Koppelman, "Memory hierarchy performance measurement of commercial dual-core desktop processors," *J. Syst. Archit.*, vol. 54, no. 8, pp. 816–828, 2008.

[40] R. Huggahalli, R. Iyer, and S. Tetrick, "Direct cache access for high bandwidth network I/O," in *ACM SIGARCH Computer Architecture News*, 2005, vol. 33, no. 2, pp. 50–59.

[41] H. Montaner, F. Silla, H. Froning, and J. Duato, "Getting rid of coherency overhead for memory-hungry applications," in *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, 2010, pp. 48–57.

[42] Barroso, L.A., K. Gharachorloo, and E. Bugnion. 1998. "Memory System Characterization of Commercial Workloads." *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*.

[43] P. Foglia, R. Giorgi, and C. A. Prete, "Simulation study of memory performance of SMP multiprocessors running a TPC-W workload," in *Computers and Digital Techniques, IEE Proceedings-*, 2004, vol. 151, no. 2, pp. 93–109.

[44] P. Foglia, R. Giorgi, and C. A. Prete, "Performance analysis of electronic commerce multiprocessor server," in *System Sciences, 2000. Proceedings of the 33rd Annual Hawaii International Conference on*, 2000, p. 9–pp.

[45] C. A. Prete, G. Prina, and L. Ricciardi, “A trace-driven simulator for performance evaluation of cache-based multiprocessor systems,” *Parallel Distrib. Syst. IEEE Trans.*, vol. 6, no. 9, pp. 915–929, 1995.

[46] J. S. Vetter, S. Lee, D. Li, and G. Marin, “Quantifying Architectural Requirements of Contemporary Extreme-Scale Scientific Applications,” in *International Workshop on Performance Modeling, Benchmarking and Simulation of HPC Systems (PMBS13)*.

[47] F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A. R. Choudhury, and Y. Sabharwal, “Breaking the speed and scalability barriers for graph exploration on distributed-memory machines,” in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, 2012, pp. 1–12.

[48] J. B. Angela, A. M. Floresb, J. S. Heritagec, N. C. Wardripd, A. M. Raime, M. K. Gobberte, R. C. Murphyf, and D. J. Mountaing, “The Graph 500 Benchmark on a Medium-Size Distributed-Memory Cluster with High-Performance Interconnect.”

[49] Z. Cui, L. Chen, M. Chen, Y. Bao, Y. Huang, and H. Lv, “Evaluation and Optimization of Breadth-First Search on NUMA Cluster,” in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, 2012, pp. 438–448.

[50] Y. Yasui, K. Fujisawa, and K. Goto, “NUMA-optimized parallel breadth-first search on multicore single-node system,” in *Big Data, 2013 IEEE International Conference on*, 2013, pp. 394–402.

[51] M. Elfituri, J. Cook, and J. Cook, “Binary instrumentation support for measuring performance in OpenMP programs,” in *Software Engineering for Computational Science and Engineering (SE-CSE), 2013 5th International Workshop on*, 2013, pp. 19–23.

[52] W. Alkohlani. “Statistical Performance Modeling Of Modern Out-Of-Order Processors Using Monte Carlo Methods.” PhD diss., NMSU, 2014.

[53] T. Austin, E. Larson, and D. Ernst, “SimpleScalar: An infrastructure for computer system modeling,” *Computer (Long. Beach. Calif).*, vol. 35, no. 2, pp. 59–67, 2002.

[54] G. H. Loh, S. Subramaniam, and Y. Xie, “Zesto: A cycle-level simulator for highly detailed microarchitecture exploration,” in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, 2009, pp. 53–64.

[55] M. T. Yourst, “PTLsim: A cycle accurate full system x86-64 microarchitectural simulator,” in *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, 2007, pp. 23–34.

[56] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, and S. Sardashti, “The gem5 simulator,” *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[57] [1] A. Patel, F. Afram, S. Chen, and K. Ghose, “MARSS: a full system

simulator for multicore x86 CPUs," in *Proceedings of the 48th Design Automation Conference*, 2011, pp. 1050–1055.

[58] A. Patel, F. Afram, H. Zeng, and K. Ghose, "MARSSx86-micro-architectural and system simulator for x86-based systems."

[59] M. E. Thomadakis, "The architecture of the Nehalem processor and Nehalem-EP smp platforms," *Resource*, vol. 3, p. 2, 2011.

[60] M. E. Thomadakis. A Westmere Addition to a High-Performance Nehalem iDataPlex Cluster and DDN S2A9900 Storage for Texas A&M University, 2011. [Online]. Available: http://sc.tamu.edu/systems/eos/Westmere-iDP.php

[61] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray User's Gr.*, 2010.

[62] D. Bader, K. Madduri, J. Gilbert, V. Shah, J. Kepner, T. Meuse, and A. Krishnamurthy, "Designing scalable synthetic compact applications for benchmarking high productivity computing systems," *Cyberinfrastructure Technol. Watch (Nov. 2006)*, 2006.

[63] M. J. Quinn, *Parallel Programming*, vol. 526. TMH CSE, 2003.

[64] D. Levinthal, "Performance analysis guide for intel core(TM) i7 processor and intel xeon(TM) 5500 processors," *Intel Corporation, Tech. Rep.* Version 1.0, 2009.