MODEL, PREDICT, AND MITIGATE SCALABILITY BOTTLENECKS FOR PARALLEL APPLICATION ON MANY-CORE PROCESSORS

A Dissertation
Presented to
The Academic Faculty

By

Ching-Kai Liang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

August 2018

MODEL, PREDICT, AND MITIGATE SCALABILITY BOTTLENECKS FOR PARALLEL APPLICATION ON MANY-CORE PROCESSORS

Approved by:

Dr. Milos Prvulovic
School of Computer Science
Georgia Institute of Technology

Dr. Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Dr. Moinuddin K. Qureshi
School of Computer Science
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Christopher J. Hughes
Intel Research
Intel

Date Approved: July 16, 2018

To my beloved wife and our first daughter whom I have received endless support and love, and to my parents and sister whom I given me strength and guidance to overcome any obstacle in life

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation and gratitude to my advisor, Dr. Milos Prvulovic. Without his vision, encouragement, and advice, I would not have been possible to finish this journey. From our first work on synchronization accelerator, to writing the thesis at the end, he constantly providing insightful feedback and directly pointed out any improvements I needed. I would also like to thank my committee members, Dr. Sudhakar Yalamanchili, Dr. Hyesoon Kim, Dr. Moinuddin Qureshi, and Dr. Christopher Hughes for their support and valuable input in improving my thesis.

My Ph.D would also not be successfully finished without the great support and encouragement from my family. My wife, Yi-Ting Tsai, has supported me throughout this journey, even gave birth to our daughter, Kyra. She is marvelously wife and mother and shouldered many responsibilities to relive my pressure and let me concentrate on my PhD. My mother, Sue Wang, whom have given me the greatest love and care any child can ask for and supported my decision to get a PhD degree. My sister, Wendy Liang, whom have pushed me and challenged me to bring out the best in me, and ensured I kept pushing toward the finish line and never give up.

I would also like to thank many friends and colleague. Ioannis Doudalis, Jungju Oh, and Anshuman Goswami, who are senior members of my lab and gave me many helps along the way. Sunjae Park, whom we road side-by-side along the PhD journey, gave me great support for not just work in school, but also navigating the new life as a parent. Chia-Chen Chou, whom reviewed many of my paper submissions and draft run presentations, helped me improve my craft work as an effective researcher. Yu-Shan Lin, whom was a great friend stood by my side during my first few years. Christine Lin, let me stay with her for five years and was a family to me when I first arrived in Atlanta.

Last but not the least, I would like to thank the many people whom have helped me and shaped me to this stage in my life. Many teachers and friends have taught me how

to navigate the difficulties in life and how to accomplish my dream by hard work and perseverance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

In recent years, the number of processor cores on a single chip has increased rapidly, ranging from hundreds of cores in server processors to tens of cores on mobile processors. The abundant number of processing cores have led to application developers investing in parallelizing applications in order to extract the maximum performance from many-core processors. However, ensuring the continuous scaling of parallel applications is challenging on many-core processors, due to the complex relationship of available parallelism in application and the limited shared on-chip resources.

Two main bottlenecks that limit the scalability of parallel applications are synchronization and memory bandwidth. Synchronization increases with the number of threads, due to the high lock contention from threads accessing the same lock-protected data that causes increase lock contention, whereas barrier operations ensure each parallel thread are within the same computation phase which limits the available thread-level parallelism. The increase in number of threads also puts more pressure on the memory subsystem in order to provide sufficient memory bandwidth for each active thread.

With this thesis, I propose both statistical models to mitigate the bottlenecks and software/hardware solutions to improve and address the scalability bottlenecks. First, I propose MiSAR, a minimalistic synchronization accelerator (MSA) that supports all three commonly used types of synchronization (locks, barriers, and condition variables), and a novel overflow management unit (OMU) that dynamically manages its (very) limited hardware synchronization resources. The OMU allows safe and efficient dynamic transitions between using hardware (MSA) and software synchronization implementations. This allows the MSA's resources to be used only for currently-active synchronization operations, providing significant performance benefits even when the number of synchronization variables used in the program is much larger than the MSA's resources. Because it allows a safe transition between hardware and software synchronization, the OMU also facilitates

thread suspend/resume, migration, and other thread-management activities. Finally, the MSA/OMU combination decouples the instruction set support (how the program invokes hardware-supported synchronization) from the actual implementation of the accelerator, allowing different accelerators (or even wholesale removal of the accelerator) in the future without changes to OMU-compatible application or system code.

Second, I propose a new performance model that captures program characteristics of multi-threaded applications, allowing it to use few-threaded runs along with small input sets to predict performance of many-threaded runs with large input sets. First, we partition the program execution into barrier phases, and model the scaling trend of the total instruction count and its distribution among threads for each barrier phase in order to account for parallelization overheads. Second, we subdivide each barrier phase into small intervals, and model the cache miss rate of each interval by utilizing the regular shifting of concurrent reuse distance (CRD) profiles. Applying the CRD analysis to small intervals allows the CRD profile to capture behavior and model performance of each phase of the program individually, rather than trying to model the aggregate behavior of potentially many phases that may differ widely in terms of cache capacity and memory bandwidth demand. Third, we use a simplified DRAM model to capture the impact of the memory subsystem on the total execution time. Finally, we model how the number of barrier phases and the model parameters (instruction count and CRD) changes with input size to predict across different input sets.

Last, I propose a PC-based profile and modeling technique to predict the increase of lock contention when scaling the number of threads. Our lock contention model consists of 4 parts. First, we divide the program execution into parallel phases separated by global synchronization (barrier, fork-join, etc.). Second, we collect statistics that represent the synchronicity of thread arrival (lock arrival rate) as well as the functionality of the corresponding critical section (size of the critical section) for each lock PC. Third, we approximate the rates into well-known statistic models (eq. exponential distribution, gaussian

distribution, etc.) in order to reduce the parameters required to model the lock contention. Last, we use regression models to predict how the parameters will change when varying the number of locks and input size.

CHAPTER 1
INTRODUCTION

1.1 Multi-core Era and Parallel Applications

In recent years, the number of available cores in a processor is increasing rapidly while the pace of performance improvement of an individual core has been lagged. As a result, applications are now required to extract more parallelism and leverage the abundant number of cores to ensure continuous speedup of their applications. However, ensure application scale well over many threads is a challenge task, mainly because scalability bottlenecks such as synchronization will saturate the performance gain if not managed carefully. In addition, finding the optimal thread count to balance the overhead and benefit of parallelization becomes even more critical.

To tackle the challenge of ensuring the scaling of parallel applications, in this work I propose both a synchronization accelerator and performance models to address the issue. The synchronization accelerator reduces the amount of synchronization overhead, specifically handoff overhead. This would allow the efficient execution of synchronization operations such as lock and barriers, while minimizing the overhead. Second, I propose two performance models to predict the scaling trend of an application. This allows application developers to estimate the potential speedup for a given system and identify rather the application would be memory-bound, compute-bound, or synchronization-bound.

1.2 Scalability challenges

1.2.1 Synchronization operation overhead

Synchronization latency is critical for achieving scalable performance on many-core processors. Numerous hardware mechanisms for low-latency synchronization have been pro-

posed [1, 2, 3] and even used in prototype and commercial supercomputing machines [4, 5, 6, 7, 8, 9, 10]. As general-purpose processors have shifted their focus from solely increasing single-core performance to providing more cores, there has been a renewed interest in hardware support for synchronization [11, 12, 13, 14, 15], this time for a much broader range of systems, programmers, and users. Examples of recently proposed hardware synchronization mechanisms include utilizing a small buffer attached to the on-chip memory controller to perform synchronization and allow trylock support [15], incorporating a lock control unit to both the core and memory controller to allow efficient reader-writer lock [14], or leveraging low latency signal propagation over transmission lines for lock and barrier synchronization [11, 12, 13].

Because previous research on hardware supported synchronization mostly focused on how to reduce synchronization latency, most such work assumed that a sufficient amount of hardware resources is available, with only limited consideration on how to handle limited hardware resources. This, however, limits the adoption of hardware synchronization accelerators both because of high cost (many applications use a large number of synchronization addresses for which resources would be needed) and correctness (some applications can exceed the resources that were considered sufficient at hardware design time).

Most prior work tackles limited hardware resources using one of the following three mechanisms. The simplest mechanism is to have an a-priori partitioning of synchronization addresses into hardware-supported and software-supported ones. Programmers will thus use hardware synchronization instructions for some and software library calls for other synchronization variables. However, this places a heavy burden on programmers because: 1) they must decide which synchronization approach to use for which synchronization variable, 2) debug problems that occur when a synchronization variable erroneously mixes synchronization implementations, and hardware resources are oversubscribed. Another mechanism is to simply stall the synchronization operation until resources are available. Although this does not require any programmer intervention, it can result in great perfor-

mance loss, or even deadlock if resources are not sufficient. The third mechanism is to treat the insufficient hardware resources as an exception. The exception handler will then decide to wait and try again or use a software synchronization mechanism. Such fallback mechanism results in significant performance penalty, so sufficient resources are needed to keep the number of the fallbacks very low. Furthermore, naively falling back to a software implementation can break the synchronization semantics, and additional overheads (and possibly additional hardware mechanisms) are needed to prevent such problems.

Also, previous proposals have focused on only accelerating (supporting) one type of synchronization. This would result in significant hardware cost/complexity to support the overall synchronization needs of real workloads, where different applications (or even the same application) may use locks, barriers, and/or condition variables. Each hardware synchronization mechanism (e.g. one for locks and another for barriers) may have its own software interface and its own verification complexity, which complicates adoption by both hardware architects and by programmers.

Therefore, in this thesis, I propose a minimalistic synchronization accelerator (MSA). The MSA is a distributed synchronization accelerator for tile-based many-core chips. It follows the POSIX pthread synchronization semantics and supports all three common types of synchronization (locks, barriers and condition variables) but has very few entries in each tile. We also propose a novel hardware overflow management unit (OMU) to efficiently manage limited hardware synchronization resources. The OMU keeps track of synchronization addresses that are currently active in software, so we can prevent these addresses from also being handled in hardware. The OMU also enables the accelerator to rapidly allocate/deallocate hardware resources to improve utilization of its (few) entries. Finally, we propose ISA extensions for hardware synchronization. These ISA extensions facilitate adoption by allowing synchronization libraries to only be modified once (to support the new instructions) and then used with any hardware synchronization implementations that support the ISA's fallback semantics – including trivial implementations with no actual

hardware synchronization support.

1.2.2 Thread Count

With the commercialization of many-core processors, such as Intel's Xeon Phi [16] and Tilera [17], a single-node system can easily exceed hundreds of processing cores. This raises a critical question of how many cores/threads to use in order to obtain close-to-optimal performance for a particular application. Prior work has shown that simply using the same number of threads as cores may not yield the optimal performance, since increases in parallelization overheads can exceed performance gains from additional parallelism [18, 19].

One approach to predict performance scaling is to explore different system configurations using detailed simulators, then use regression methods to build a statistical predictor [20]. However, training a regression model for accurate results require thousands of data points, which is time consuming and therefore cannot quickly identify the scalability trends of an application. Others have proposed methods that dynamically manage the number of active threads at runtime [19, 18], typically by starting with few threads, predicting the thread count that will saturate an available resource (eq. memory bandwidth), and adjusting the thread count accordingly. However, these prediction mechanisms often assume that each thread has identical working-set size and program behavior[19], which does not take into account load imbalance. Furthermore, most dynamic approaches can only provide performance prediction up to the number of available cores, which precludes their use for estimating performance gains that might be obtained with higher-core-count processors (e.g. to determine if performance gains would justify upgrading the processor to a higher-core-count one).

Profiling tools have also been used for identifying potential scalability bottlenecks [21, 22], typically by utilizing hardware performance counters to account for various microarchitectural events such as last level cache (LLC) misses, high-latency instructions, etc., and

then components of the execution time that are particularly detrimental to scaling. Such profiling approaches are typically very helpful in situations where the profiled execution is already experiencing substantial performance degradation, but do not provide much insight into how performance will *eventually* degrade at significantly higher thread counts.

For applications that scale well, the total amount of work and its distribution among threads becomes the dominant factor for efficient scaling. Most prior work assumes a constant total instruction count as the thread count increases[23, 19, 18], i.e. that there is no per-thread work that must be done by each thread regardless of how many threads are used. However, even a small amount of per-thread work becomes important when using many threads. For example, Figure 1.1 shows how the total instruction count scales for the Splash-2 benchmark *Cholesky*. While the total instruction count is close-to-constant for low thread counts, it rapidly increases when using many threads, so that a 128-thread execution executes 9X the number of instructions executed by the single-threaded execution. This means that modeling of the per-thread overheads is very important when trying to predict performance scaling of some applications. Figure 1.1 also shows the ratio between the maximum and average instruction count among threads, and also the maximum-to-minimum per-thread instruction count. We observe that this ratio increases as the thread count increases, i.e. at higher thread counts the application will experience higher load imbalance. Predicting this load-imbalance can become important when trying to predict overall application performance at higher thread counts.

One challenge for using statistical models to predict performance is the complexity of the model and the required number of data points. Naively predicting the program characteristic would require large amount of data points in order to correctly capture the complex program behavior. However, prior work have shown that synchronization barriers are a natural boundary for a program phases, and consists of homogeneous behaviors across and within barrier phases [24, 25]. Our work extends upon this concept and shows that barrier phase characteristic are also homogeneous across different input, and by leveraging

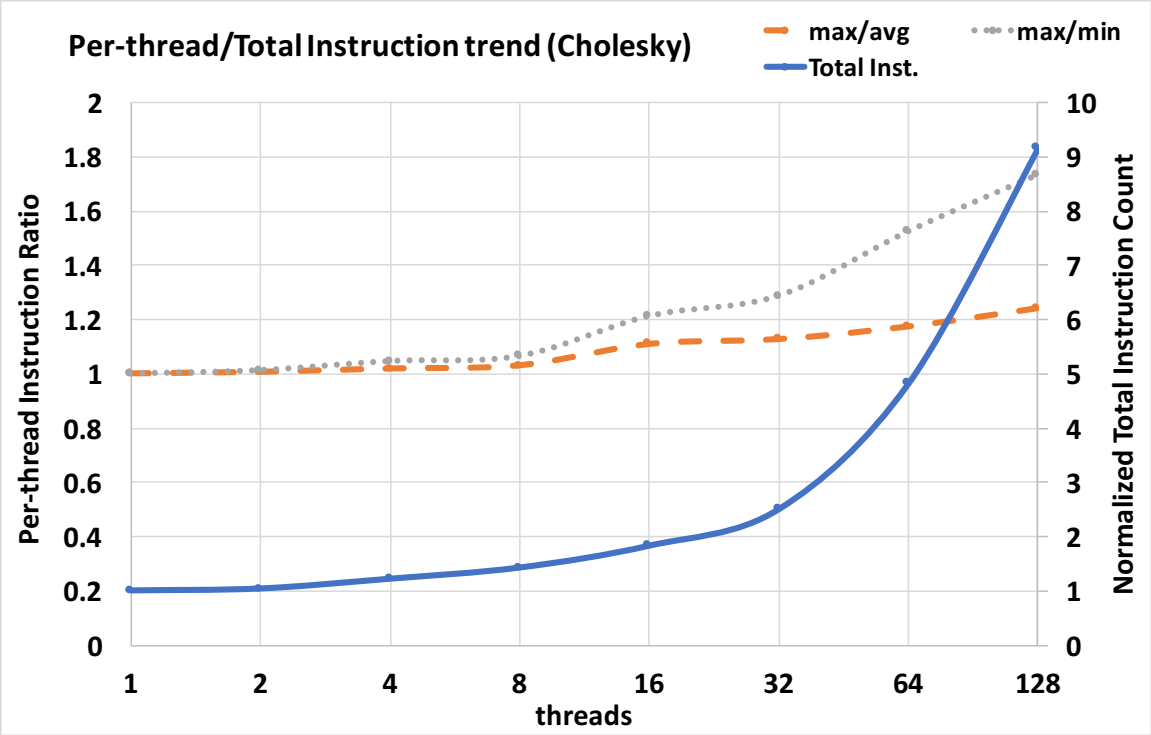

Figure 1.1: Per-thread/Total instruction trend for Cholesky

such program structure, we can not only reduce the complexity and required data points to achieve better accuracy for modeling program characteristics, but also predict how program scales across larger input sets. Figure 1.2 shows how the accuracy of predicting the total instruction improves by subdividing the prediction on each barrier phase verses over the whole program.

Contention for memory resource is another main bottleneck for application scalability. Increasing the number of active threads increases the demand for data movement bandwidth (both in the on-chip interconnect and in the memory channels), degrades cache locality in shared caches (which further increases memory bandwidth demand), and often increases the total memory footprint by increasing the total amount of tread-private data (which additionally degrades cache performance and results in even more memory bandwidth pressure). Several prior works have studied how thread-count degrades memory locality [26, 27] and increases pressure for off-chip memory bandwidth [21]. However, these works only target a single architecture feature (cache miss rate, memory bandwidth utilization), and do

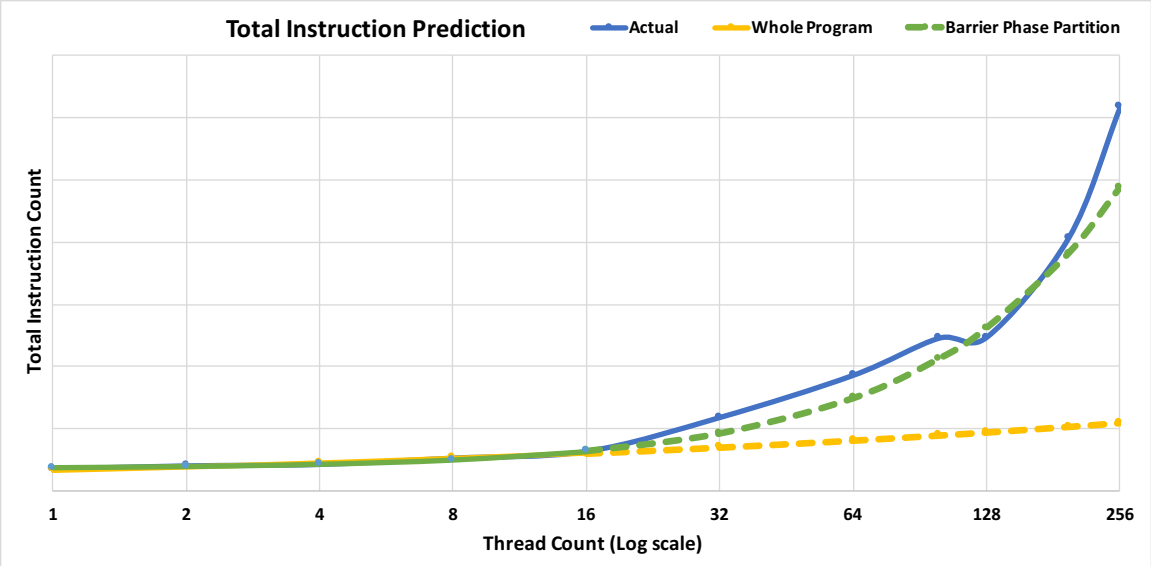

Figure 1.2: Instruction Prediction

not address the problem of predicting overall performance scaling in general, especially when multiple performance-limiting factors are present and their effects are compounded.

To illustrate how one factor is unlikely to provide a good picture of overall parallel performance, Figure 1.3 shows the overall parallel speedup and MPKI of Splash-2 [28] benchmark *FFT* as the number of threads increases. As shown, MPKI increases by 2X when scaling from 1 thread to 256 threads. However, the parallel speedup peaks at 64 threads, which is beyond the point where MPKI starts to degrade. Beyond 64 threads the MPKI continues to increase, while the parallel speedup degrades. This speedup degradation is actually caused by compounding the effects of cache hit rate degradation (which is accounted for in the MPKI) and the saturation of memory bandwidth (which is not accounted for in the MPKI). Note that cache hit rate degradation alone would be expected to only reduce the slope of the performance growth curve, while the bandwidth saturation alone would be expected to cause saturation of the parallel speedup (making it asymptotically approach a constant value). However, with both effects present simultaneously, bandwidth saturation prevents any parallelism-related gains from thread-count increases while cache hit rate degradation causes each thread to slow down, resulting in degradation

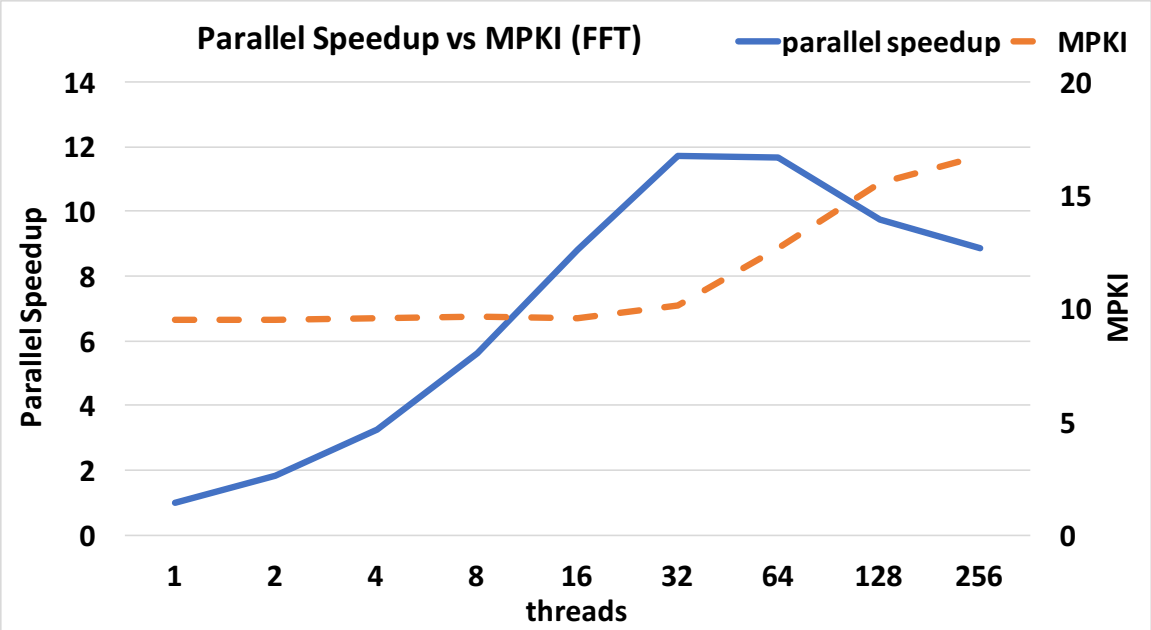

Figure 1.3: Parallel Speedup vs. MPKI

in performance as the thread-count increases.

To predict the interaction between cache hit/miss performance and the limited bandwidth in the memory system, we adopt a mechanism proposed by M.-J. Wu et al. [26] that uses an augmented Reuse Distance (RD) analysis to better account for how cache hit/miss performance changes with the thread count. In addition, we also account for bandwidth-related considerations (memory access burstiness) and instruction-count trends to model and predict the interaction among these performance-limiting factors. We note that RD analysis was originally applied to sequential programs, and that several works have extended RD analysis to analyze how core-count scaling affects multi-program workloads [29] and parallel programs [30]. However, these studies mainly focus on how the total number of cache miss changes but neglect the changes in the burstiness of cache misses and the overall parallel speedup.

In result, I propose a performance prediction model that consists of 4 main components. First, I partition the program execution into barrier phases, and model the scaling trend of the total instruction count and its distribution among threads for each barrier phase in order

to account for parallelization overheads. This identifies how the total work changes and how work distribution among threads changes as the thread count increases. Second, we predict the cache miss rate at small intervals by utilizing regular shifting of concurrent reuse distance (CRD) profiles. By applying the CRD analysis to small intervals, we could associate CRD profile with program phases and capture bursts of memory requests. Third, we use a simplified DRAM model to capture the memory subsystem slowdown and its effect on the total execution time. Last, we model how the number of barrier phases and the model parameters (instruction count and CRD) changes with input size to predict across different input sets.

1.2.3 Lock Contention

Many-core processors, some with 10s or even 100s of processing cores, have become ubiquitous in all sections of computing, ranging from handheld mobile devices [31], to accelerators [16, 17]. As many-core processors become ubiquitous, parallel applications are also gaining increasingly prevalent as developers and users desire to fully utilize the abundant processing power available across the cores in a single system. However, as the number of cores increases, developers and users expect the performance to scale, i.e. the amount of useful computation achieved per unit time should increase with the number of cores used for that computation. Unfortunately, good performance scaling is difficult to achieve in practice, mainly because of various bottlenecks that can each limit performance scaling. One of the common performance scaling bottlenecks is lock contention, which effectively forces serialization of execution and thus prevents all the cores from concurrently doing useful computation.

Prior work have proposed methods to combat the increase of lock contention when scaling applications. Hardware solutions have been proposed to reduce the overhead for performing lock operations, such as a dedicated hardware accelerator for course-grain [32] or fine-grain [15] locking, or micro-architectural features which predicts and opportunisti-

cally executes the critical section in order to reduce serialization [33]. Software solutions such as runtime systems to change the scheduling policy have also been proposed to reduce the contention on lock variables [34][35][36]. However, these approaches mainly focuses on reacting to the serialization of lock contention and does not identify or address the fundamental problem in the program.

Various performance analysis and profiling techniques have been proposed in order to identify and remove the fundamental problem of serialization in the original program. Chen and Stenstrom proposed a mechanism to identify the longest critical path from an execution trace file in order to attribute the cause of the scaling bottleneck [37]. Bois et al. proposed a new criticality metric in order to capture the severity of lock serialization [38]. Others have also proposed light weight mechanisms to identify lock contention [39] and to categorize various type of bad synchronization [40]. Commercially available tools such as Intel vTune[41] and Concurrency Visualizer in Microsoft Visual Studio [42] allows programmers to debug and identify performance bottlenecks in the system. They do not, however, provide insight into if or when a bottleneck would occur in other runs, e.g. if more cores were used.

Analytical models have also been proposed to model lock contention in various systems. Yu et al. [43] proposed a database performance model that assumes a Poisson arrival rate of transactions, with each transaction accessing a set of locks what have uniform access probability. Thomasian [44] generalized such model to incorporate multiple class of transactions, each with different distribution of lock access probability. However, database systems fundamentally differ from homogeneous multi-threaded applications in that, for database systems, transactions arrive independently of each other as a result of external requests, so their arrival rate which can be modeled as a Poisson distribution and their lock contention is a good match for a traditional queuing model. In contrast, in multi-threaded applications threads tend to be, at least to some degree, in lock-step with each other, and therefore threads are not necessary arriving independently. In addition, during the lifetime

of a threads, it will access various lock variables during different program phases, which behavior is different than database systems.

To provide insight into the scaling of multithreaded applications, Eyerman and Eeckhout [45] proposed extending Amdahl's law to model the increase in serialization due to scaling. However, the model simply assumes random arrivals to critical sections, which prevents the model from providing accurate quantitative performance predictions. Boyd-Wickizer et al. [46] proposed a lock contention model for ticket locks by using Markov models in order to calculate the expected number of idle cores. However, their model only considers a single lock, with no discussion on how the model can be applied to applications with multiple locks.

In general, one of the main drawbacks of prior lock contention model is the assumption of uniform and random access of lock variables during the whole application. However, these assumptions are violated by many real programs, e.g. because different program execution phases can have very different lock-related behaviors. To illustrate this, Figure 1.4 shows the number of lock access over time for *Radiosity*. As shown, the lock accesses are more prevalent early in the application and, when lock accesses do occur later in the application, they are clustered together (the peaks at several points in the execution timeline).

To model and predict the lock contention of applications, we propose to model the lock contention separately for each static location in the code (PC address) at which the lock is acquired. Our intuition is that the overall lock-related behavior of the application is a combination of behaviors in different program phases. Furthermore, even within each program phase the overall behavior can be a combination of different behaviors as different data structures may be protected by different sets of locks that can substantially differ in locking behavior. Intuitively, we expect that each of these individual behaviors can be modeled more simply and accurately than the entire application, and that these individual models can be combined in a relatively straightforward way into a model for the entire application that will be more accurate than a model that considers the entire application

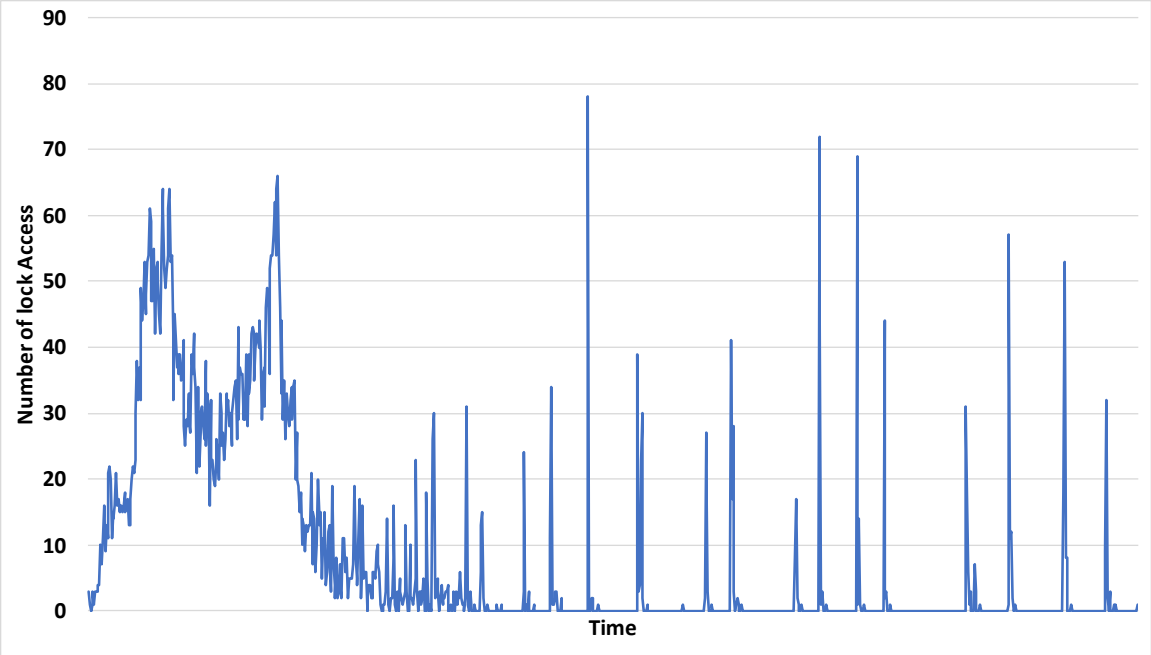

Figure 1.4: Number of lock accesses over 1M cycles (1K data points) for 128-thread execution of Radiosity

as a homogeneous whole. Since the PC of the lock operation is correlated with both the current program phase and with the set of locks that is used, PC-specific modeling of lock behavior should help capture these individual behaviors.

Additionally, we observe that after global synchronization, such as a barrier, threads tend to have a high degree of synchronicity, i.e. they are close to being in lock-step with each other, and this makes lock contention much more likely than when threads arrive to critical sections randomly.

Figure 1.5 shows the lock access over time for each lock PC. As shown, we can see that during the first peak of lock access, the "'red" PC (PC1) is the dominant source of lock accesses, while later in the execution the "green" PC (PC2) is dominant. This is because of during a single parallel phase, threads execute in different regions of code, therefore exhibit different phase behavior. Furthermore, the "green" lock accesses come in bursts, which is the result of threads executing in relatively similar code regions, therefore would access the same lockPC in synchronicity. This shows that lock PC is a useful proxy to capture to

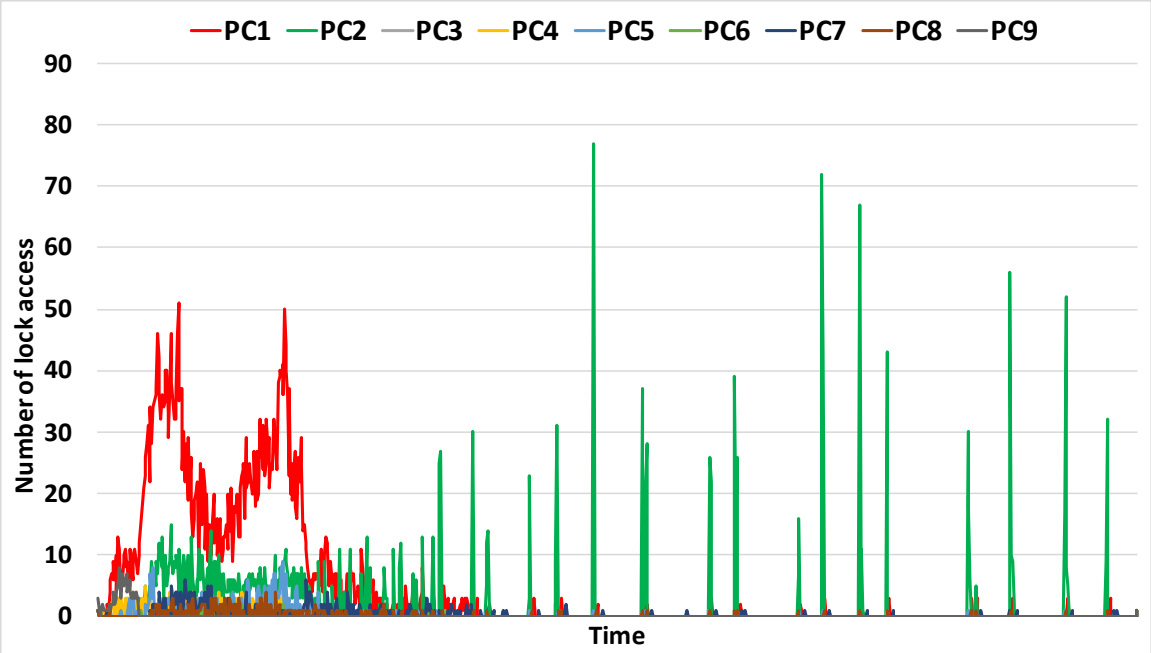

Figure 1.5: Per-PC Lock access over time

phases of thread's lock access. In addition, the characteristic of each lockPC also displays a structural change, such that it allows us better to predict the arrival rate or lock histogram.

In summary, our lock contention model consists of 4 parts. First, we divide the program execution into parallel phases separated by global synchronization (barrier, fork-join, etc.). Second, we collect statistics that represent the synchronicity of thread arrival (lock arrival rate) as well as the functionality of the corresponding critical section (size of the critical section) for each lock PC. Third, we approximate the rates into well-known statistic models (eq. exponential distribution, gaussian distribution, etc.) in order to reduce the parameters required to model the lock contention. Last, we use regression models to predict how the parameters will change when varying the number of locks and input size.

1.3 Thesis Statement

With the accelerating technology improvements, the number of available cores in a processor steadily increases. As a result, it is necessary for application developers to exploit parallelism for better application performance. However, mitigating the parallelization

overhead while also determining the optimal thread count is a challenging task. Various scalability bottleneck may occur when scaling application on many-core processors, such as limited parallelism in the application design, poor synchronization algorithm, or even limited memory bandwidth to support all the necessary data movements from all the active threads.

This thesis proposes a performance model to understand the potential scaling trend for a given application, as well as a hardware accelerator to mitigate the scaling bottleneck. The thesis proposes the following statement: **Statistical models and hardware techniques can help understand and improve the scaling of parallel applications on many-core processors**

1.4 Thesis Overview

Chapter 2 provides the background and related works on synchronization accelerator design, performance modeling techniques, and lock contention analysis methods. Chapter 3 explains MiSAR, a hardware synchronization accelerator that reduces the synchronization overhead. Then, in Chapter 4, a new statistical model technique is proposed to model how application scales with thread count, and determine the optimal thread count for maximum parallel speedup. Chapter 5 discusses a PC-based statistical lock contention model in order to model and predict how lock contention scales with thread count. I conclude in Chapter 6.

CHAPTER 2
BACKGROUND AND RELATED WORK

2.1 Synchronization Accelerators

Hardware support for synchronization generally improves synchronization in two ways. First, by implementing the synchronization semantics in hardware, which avoids the inefficiencies in updating the synchronization state in software. Examples include accelerators for barriers [47, 48, 49] that track barrier's arrival state and detect the all-arrived condition without the overhead of updating the arrival count variable in a critical section. Lock accelerators [50, 47, 49, 15] maintain the lock's owned/free state in hardware and thus help arbitrate which requestor is the next to get the lock once it is freed.

The other way is to improve synchronization latency by directly notifying the waiting threads to avoid the coherence "ping-pong" involved in software-only synchronization. For example, a software-only implementation of lock handoff involves sending an invalidation when releasing the lock, waiting for that invalidation to reach all the sharers (typically all cores waiting for that lock), a cache read miss on (at least) the next-to-acquire core, a transfer of the lock's block into that cache, and then sending an upgrade request (invalidation) when actually acquiring the lock. In contrast, a direct-notification lock accelerator [12, 51, 2, 14] typically involves sending a single message from the releasor to the next-acquirer. A similar flurry of coherence activity is involved in signaling barrier release in software, and is avoided in hardware accelerators [11, 1, 13] by directly signaling the barrier release to waiting cores.

Synchronization support has also been used in distributed supercomputer systems, e.g., efficient broadcast networks have been used to accelerate barrier operations [9, 52], and fetch-and-add operations have been used for efficient barrier counting [5, 7, 10]. Our work

focuses on more tightly coupled many-core systems, and provides support for all three common types of synchronization.

In addition to improving synchronization latency, most hardware synchronization accelerators also have to handle what happens when hardware resources of the accelerator are exhausted. Several solutions have been adopted in prior work, such as requiring programmers to manually partition synchronization variables [11, 51, 1, 50, 13] into those that always use software and those that always use hardware, using the memory as a resource buffer [14], or switching to a software exception handler [15]. Unfortunately, programmer-implemented partitioning is not portable to architectures that have fewer resources, use of main memory complicates the implementation and adds latency, and software exception handlers are difficult to implement correctly and can incur large overhead when fallbacks are too frequent. Utilizing the memory as a resource buffer can reduce the amount of software exception events [14], however, still requires the exception handler to resize the resource table. In addition, going to main memory to access the resource buffer increases the overall synchronization latency. In contrast, our approach uses a small OMU local to each tile to efficiently and correctly fall back to an existing (e.g. pthreads) software implementation when needed, while also improving utilization of the (very limited) hardware synchronization resources. A more detail discussion of our scheme verses software exception handler will be discussed in Section 3.1.2.

Table 2.2 summarizes the prior proposals for multi-core synchronization: which synchronization types they support, whether they provide direct notification, the hardware cost (in terms of added state), whether a specialized network is required, and how hardware resources overflow is managed. For resource overflow, `SW` corresponds to simply falling back to a software handler when resources are exhausted, whereas `HW` resolves it in hardware. For LCU [14], it will first fallback to the memory and only if memory overflows will it require a software handler, thus we mark it `HW/SW`.

In general, accelerators that provide direct notification support only one type of syn-

chronization (e.g. only lock or only barrier), and direct-notification barrier proposals mostly rely on dedicated networks. Also, many of the mechanisms require recording state information that is proportional to the number of locks or barriers in the system - potentially many different locations, especially for programs that use large arrays of locks. In addition, so far no barrier accelerator has tackled the problem of resource overflow. In contrast, our proposed approach supports all three types of synchronization (locks, barriers, and condition variables), with direct notification over the existing on-chip interconnect, and with $O(N_{core})$ hardware resource overhead.

2.2 Performance Modeling

Various regression modeling techniques have been applied to performance modeling. B.C. Lee et al.[20] applied regression modeling technique to develop a non-linear model for reducing the work of design space exploration. The proposed regression model uses 4000 sample points to derive an architectural-application predictor. B.J. Barnes et al. [53] also uses regression modeling to predict the scalability of applications. This model separates computation and communication and then fits the data points into a linear regression model. More general regression models, such as artificial neural networks, were applied by E. Ipek et al. [54] for performance prediction. However, these models do not directly attribute the degradation in speedup to the factors that cause it (per-thread overhead and imbalance, cache hit/miss degradation, and bandwidth limits) nor provide an intuitive model for how these factors interact to produce the overall performance trend.

In addition to regression models, optimization techniques have also been applied to performance modeling. W. Wang et al. [55] use integer programming for optimal core/node placement for NUMA systems by collecting local and inter-node bandwidth usage, along with DRAM bandwidth and contention. Unfortunately, in such schemes the number of input parameters grows exponentially with the number of cores. In contrast, our approach leverages program characteristics to reduce the number of model parameters, uses model

parameters that can directly be used to attribute/explain which program and microarchitectural factors are responsible for degradation in application speedup, and can use parameters obtained at lower core counts to predict performance on larger core counts.

Several works have also been proposed to predict the memory subsystem performance by using a detailed analytical DRAM model. DraMon [56] predicts memory bandwidth usage by modeling memory issue rate and row buffer hit/miss/conflict ratio using probability models. The probability for co-running threads to access the same rank/bank/channel is calculated from memory traces and assume that all threads have the same probability. However, this work only predicts memory bandwidth utilization and does not account for its interaction with other factors or its impact on overall execution time. ANATOMY [57] proposed a 3-stage queueing model for memory system performance. The model assumes an arrival rate with exponential distribution and mean $1/\lambda$, with memory banks and data bus as M/D/1 queues. To model the processor performance, it utilizes the CPI stack, which assumes the ideal memory CPI and then add the penalty by the memory subsystem. In contrast, our work uses a relatively simple memory bandwidth model, but also uses a model of per-thread overheads and a model of cache hit/miss behavior to predict how these factors jointly affect overall performance of a parallel application.

Dynamic runtime systems have been proposed to determine the optimal number of threads on the fly. FDT [19] predicts the optimal number of threads by first sampling the program characteristic in serial (using only 1 thread). It assumes the program will be either memory bound or synchronization bound, and collects the time spent in critical section and memory utilization to determine if either memory or synchronization is limiting the performance scaling.

CRUST [18] predicts the performance for clustered cache architectures. By sampling the miss rate for different active-cores-per-cluster during the end of each parallel section, it is able to correlate the number of threads with cache miss rate. With such information, it predicts the optimal thread count for each cluster by calculating the number of threads

needed to saturate the total memory bandwidth.

M. Kim et al.[23] predict the potential speedup with a non-parallelized serial code That approach predicts DRAM access overheads by assuming that performance can be decoupled into computation and memory, that work is equally partitioned among threads, and that the LLC miss rate will not change when parallelizing the application.

2.2.1 RD analysis

Multicore reuse distance analysis has been applied to study the cache behavior of parallel application. Ding and Chilimbi [30] discussed the construction of concurrent reuse distance (CRD) from per-thread RD by statistically interleaving memory accesses from different threads. It requires modeling each thread's sharing behavior to statically determine the interleaving behavior of memory accesses. M.-J. Wu et al. [26] simplified CRD analysis by leveraging the symmetry of threads for loop-based parallel programs. It utilizes reference group to predict how the CRD profile shifts and scales when increasing thread count. Our work applies CRD analysis to shorter intervals, which allows it to consider not only how the cache size and thread count affect the overall cache miss rate but also how the cache misses are distributed over time and how they interact with memory bandwidth to affect the parallel performance of each interval and, by combining the resulting performance of all the intervals, of the entire application.

D. Chandra et al. [29] and G.E. Suh et al. [58] looked at how CRD analysis can be applied to multi-programing workloads. While our work mostly considers parallel applications, some of our insights, e.g. combining CRD analysis with considerations for cache miss burstiness and memory bandwidth, may be applicable in that domain.

2.2.2 Tools for Analysis of Performance Scaling

Many tools have been developed to identify scaling bottlenecks in multi-threaded applications. S. Eyerman et al. [22] proposed using speedup stack and breakup the performance

slowdown into synchronization and resource sharing. W. Heirman et al. [59] proposed using cycle stack to attribute which microarchitecture structure is limiting parallel scaling.

Several other commercial profiling tools, such as Intel VTune™ [41] or HPCToolkit [60], identifies scalability bottlenecks by profiling the application's execution. Although these tools offer scaling bottleneck analysis beyond just the memory subsystem, they require executing the application on the target system with the target thread count to identify the scalability bottleneck(s). Hence, does not provide prediction mechanism on when a bottleneck would occur when scaling up thread count.

In summary, Table 2.2 summarizes the prior work on predicting the performance scaling of a parallel program.

2.3 Lock Contention Analysis

2.3.1 HW/SW lock contention reduction mechanism

Various hardware/software mechanisms have been proposed to reduce the severity of lock contention. Hardware solutions such as MiSAR [32] and SSB [15] utilizes on-chip hardware accelerators to reduce the overhead for performing lock operations. These proposals rely on implementing the lock functionality directly in hardware in order to improve the lock access latency. Other proposes such as Lock Elision [33] utilizes speculation techniques to optimistically execute critical sections concurrently, even when the critical sections are protected by the same lock variable. In case of mis-speculation, rollback is performed to ensure the correct execution of conflicting critical sections.

Software techniques have also been proposed to reduce the severity of lock contention based on runtime systems to dynamically controlling the number of active threads. FDT [34] samples the time spent in a critical section for a single threads, and then assumes each thread spends the same amount of time in critical section for each loop iteration, and schedule enough threads to fill the loop iteration with critical sections. Sridharan et al. [35] utilizes a modified Amdahl's law model to take into account serialization caused by lock,

and then sample the execution runtime with various degree of threads in order to obtain the parameters for the modified Amdahl's law. Cui et al. [36] monitors the time spend on waiting for locks, and use empirical data to determine a threshold for determining when to stop scheduling more threads.

In summary, these proposals focus on reacting to the increase of lock contention and try to reduce the severity of such contention either through microarchitecture features to accelerator lock operations, or simply reduce the amount of active threads. Note, however, they do not provide a means to predict when a contention will happen, nor do they reveal why such contention exists.

2.3.2 Lock profiling tools

Various performance analysis and profiling techniques have been proposed in order to identify and remove the fundamental problem of serialization in the original program. Chen and Stenstrom [37] proposed using critical lock analysis to identify the longest critical path in order to contribute the cause of the scaling bottleneck. They proposed mechanism first collects the trace of all lock events, and then calculating backwards from the end of program execution to determine the longest critical path. Tallent et al. [39] discussed how to profile and attribute performance loss due to lock contention by recording the number of waiting threads when a lock is released, and attributing the lock holder for the idleness of each waiting threads.

Bois et al. [38] proposed a new criticality metric (Criticality stack) in order to model the severity of lock serialization. The criticality of each thread is calculated as the ratio of how many work is done verses how many threads are idle. The more threads are waiting, the more critical a thread is. Hence, the criticality stack can represent the criticality of each thread and identify the possibility of improving performance by accelerating the most critical thread. Alam et al. [40], categorizes lock usage in various applications, and identify why some lock are creating high contention verses the others. They also proposed methods

to improve the locking scheme in order to reduce lock contention, such as using atomic instructions for critical sections that only contain simple integer operations.

Commercially available tools such as Intel vTune [41] and Concurrency Visualizer in Microsoft Visual Studio [42] also allows programmers to debug and identify performance bottlenecks in the system. These tools profile the runtime behavior, and reports statistics such as how many active threads are concurrently running, what do the lock wait time, etc., in order to let the programmer better understand the scaling of the application. However, these tools mainly focus on identifying the bottleneck when it actually occurs and does not provide insight into if or when a bottleneck would occur.

2.3.3 Lock contention models

Very few works have tackled the problem of modeling and predicting the scaling of multithreaded applications. Boyd-Wickizer [46] proposed a lock contention model for ticket locks by using Markov models in order to calculate the expected number of idle cores. However, their model only looked at a single lock, and does not discuss how the model can be applied to applications with multiple locks. In addition, their model also did not address how to predict the lock contention when scaling the number of threads.

Eyerman and Eeckhout [45] extended the Amdahl's law model to include the increase in serialization due to lock contention while scaling the number of threads. Their model assumes the execution runtime is either limited by the slowest thread, or the average behavior of all threads, and take the slowest prediction of the two. Since their model focus more on the general scaling behavior of an application, it assumes critical sections happen randomly and uniformly across the whole program execution. In result, their work cannot be used as an accurate quantitative performance prediction.

2.3.4 Lock contention modeling for database systems

In addition to multithreaded applications, prior works have also studied how to model lock contention for database and transactional memory systems. Yu et al. [43] proposed a database performance model that assumes a Poisson arrival rate of transactions, with each transaction accessing a set of locks that have uniform access probability. Thomasian [44] generalized such model to incorporate multiple class of transactions, each with different distribution of lock access probability. However, database systems differ from homogeneous multi-threaded applications in one fundamental way such that for database systems, transactions are independent of each other. This results in a arrival rate which can be modeled as a Poisson distribution, and that lock contention is can be modeled with a queueing model. On the other hand, multi-threaded applications tend to execute in lock steps, therefore threads are not necessary arriving independently. In addition, during the lifetime of a threads, it will access various lock variables during different program phases, which behavior is different than database systems.

Xiao et al. [62] discussed how to use a queuing model to analyze the performance for transactional memory systems. Their model assumes transaction arrive and commits according to a linear model, with the probability increases with time. In addition, their model also considers a transaction can abort and retry when a conflict is detected. A conflict is detected when two threads access the same data element, which assumes a uniform distribution of access probability of all data elements. While their works focus on a detail analytical model of a transactional memory system, their model requires many different parameters in order to properly model the system behavior, thus increases the complexity of using it as a performance prediction model when the parameters are unknown.

Table 2.1: Summary of hardware synchronization approaches

Work	**Synchronization Primitives**	**Notification**	**Resource overhead**	**Dedicated Network**	**Resource Overflow**
Lock Table[50]	Lock	Indirect	$O(N_{lock})$	No	SW
AMO[49]	Lock, Barrier	Indirect	0	No	N/A
Tagged Memory[47]	Lock, Barrier	Indirect	$O(N_{mem})$	No	N/A
QOLB[2]	Lock	Direct	$O(N_{core})$	No	SW
SSB[15]	Lock	Indirect	$O(N_{activeLock})$	No	SW
LCU[14]	Lock	Direct	$O(N_{core})$	No	HW/SW
barrierFilter[48]	Barrier	Indirect	$O(N_{barrier})$	No	Stall
Lock Cache[51]	Lock	Direct	$O(N_{lock}N_{core})$	Yes	Stall
GLocks[12]	Lock	Direct	$O(N_{lock})$	Yes	None
bitwiseAND/NOR[1]	Barrier	Direct	$O(N_{barrier})$	Yes	None
GBarrier[11]	Barrier	Direct	$O(N_{barrier})$	Yes	None
TLSync[13]	Barrier	Direct	$O(N_{barrier})$	Yes	None
MSA/OMU (Our proposal)	**Lock, Barrier, Condition Variable**	**Direct**	$O(N_{core})$	**No**	**HW**

Table 2.2: Summary of scalability prediction schemes

Work	**Inst/Work**	**Cache**	**DRAM**	**Data points**	**Predictability**
ANATOMY[57]	N/A	N/A	Arrival rate, row-buffer-hit rate bank level parallelism request spread	Memory trace	Change N_{Banks}
Cilkview[61]	Constant	N/A	N/A	Call graph	Upper/ lower bound
Linear Regression[20]	N/A	N/A	N/A	4000 points	Predict
Integer programming[55]	N/A	N/A	N/A	N_{core}^2	N
Parallel Prophet[23]	Constant	Constant (miss rate)	Constant latency	1 (Serial run) (Anotated program)	Up to 12-thread
FDT[19]	Constant	Constant (miss rate/thread)	N/A	Several loop iteration (Bandwidth utilization)	Up to 32-threads
CRUST[18]	N/A N/A	Sampled (Miss rate)	Sampled (Avg. Latency)	$N_{core/cluster}$	$N_{clusters}$ * $N_{core/cluster}$
OurProposal	Regression	RD Analysis	Linear model	5 points (1-16 threads)	up to 256 thread (16X)

CHAPTER 3

MISAR: MINIMALISTIC SYNCHRONIZATION ACCELERATOR WITH RESOURCE OVERFLOW MANAGEMENT

In this chapter, I propose MiSAR, a minimalistic synchronization accelerator (MSA). The MSA is a distributed synchronization accelerator for tile-based many-core chips. It follows the POSIX pthread synchronization semantics and supports all three common types of synchronization (locks, barriers and condition variables) but has very few entries in each tile. We also propose a novel hardware overflow management unit (OMU) to efficiently manage limited hardware synchronization resources. The OMU keeps track of synchronization addresses that are currently active in software, so we can prevent these addresses from also being handled in hardware. The OMU also enables the accelerator to rapidly allocate/deallocate hardware resources to improve utilization of its (few) entries. Finally, we propose ISA extensions for hardware synchronization. These ISA extensions facilitate adoption by allowing synchronization libraries to only be modified once (to support the new instructions) and then used with any hardware synchronization implementations that support the ISA's fallback semantics – including trivial implementations with no actual hardware synchronization support.

3.1 Design of MiSAR

Our proposed MSA is designed for a tiled many-core chip, where each tile contains a core and its local caches, a network-on-chip (NoC) router, a slice of the last-level cache (LLC) and coherence directory, and a slice of the synchronization accelerator. However, it can be adapted for use in other settings, e.g. those with broadcast interconnects (buses), centralized instead of distributed LLCs, etc.

A single slice of our synchronization accelerator is shown in Figure 3.1. It contains a

(small) number of synchronization entries, and each entry tracks synchronization state of a single synchronization address. An entry in the MSA is a global "clearing house" for all synchronization operations for that particular address. To simplify interactions with coherence, we distribute the entire MSA according to the coherence home of the synchronization address: if an MSA entry is associated with a synchronization address, that entry must be in the LLC home tile of that synchronization address.

Each MSA entry contains the synchronization address it is associated with and a valid (V) bit. It also contains what type of synchronization it is currently used for, a bit vector (HWQueue), and an auxiliary information field. The HWQueue utilizes one bit per core to record which cores are waiting on that synchronization address, and also the lock owner in case of locks. The use of the auxiliary field depends on synchronization type, as will be explained later. Here we assume that each core runs only one thread. To support hardware multi-threading, the HWQueue would be augmented to have 1-bit per hardware thread. Note that, even with 64 cores and 2 threads per core, the overall state of a single-entry MSA would be less than 264 bits (33 bytes) in each of the 64 tiles.

Software interacts with the MSA using a set of 6 instructions, each corresponding to a synchronization operation (`LOCK`, `BARRIER`, `COND_WAIT`, etc.). Each instruction has a return value that is either `SUCCESS`, `FAIL`, or `ABORT`. The instruction returns `SUCCESS` when the synchronization operation was successfully performed, `FAIL` when the operation cannot be performed in hardware, and `ABORT` when the operation was terminated by MSA due to OS thread scheduling. A more detailed discussion of `ABORT`, and how it differs from `FAIL`, is provided in Section 3.2. To simplify integration into the processor pipeline and to simplify interaction with memory consistency, each synchronization instruction acts as a memory fence and its actual synchronization activity begins only when the instruction is the next to commit. We fully model the resulting pipeline stalls in our experiments and find that they are negligible in most applications.

3.1.1 Allocate/Deallocate MSA Entry

An MSA entry is allocated (if available) when a synchronization "acquire" request (`LOCK`, `BARRIER`, or `COND_WAIT`) is received by the home of the synchronization address. The entry is evicted when its HWQueue becomes empty, i.e. when no thread waits for or owns the lock, when the barrier is released, or when no thread waits for a condition variable. As indicated earlier, if no MSA entry is available, the MSA simply returns `FAIL`, which results in using software implementation for the synchronization operation.

The MSA does not allocate a new entry for "release" requests (`UNLOCK`, `COND_SIGNAL` and `COND_BCAST`), so they fail if a matching entry is not found. This helps ensure that, if an acquire-type operation used a software implementation (`LOCK`, `BARRIER`, or `COND_WAIT` returned `FAIL`), a release will also "default-to-software".

3.1.2 Overflow Management Unit (OMU)

The Overflow Management Unit (OMU) ensures correct synchronization semantics when an MSA entry is not available. The OMU keeps track of the synchronization addresses that currently have waiting (or lock-owning) threads in software. The OMU consists of a small set of counters indexed (without tagging) by the synchronization address. Once a thread's acquire-type synchronization operation falls back to software, the counter that corresponds to the synchronization address will be incremented. The counter is decremented when the operation completes (for locks, when the lock is released). When an acquire-type operation does not find an MSA entry, we find the OMU counter for that address to check if an MSA entry can be allocated for it (OMU counter is zero), or if the operation must be done in software to maintain correctness (OMU counter is $>$ zero). Note that this requires the entering and exiting of the synchronization operation to be visible to the OMU. For locks, the entering and exiting results in attempting `LOCK`/`UNLOCK` instructions (which `FAIL` because that lock is handled in software). For barriers and condition variables, entering is similarly exposed to hardware (a *FAIL*ed `BARRIER` or `COND_WAIT` instruction). However, barrier

and condition wait that complete in software would normally not be visible to hardware, so we add the `FINISH` instruction at the end of software barrier and condition wait to inform the OMU that the operation has completed (so it can decrement the corresponding counter).

The accelerator only grants new hardware resources (allocates an MSA entry) for an "acquire" request when there is no already "active" (waiting or lock-owning) synchronization on that address in software. To illustrate why this is necessary, consider a synchronization accelerator that has already *FAIL*ed several `LOCK` requests for a given address because MSA resources were not available. As a result, the lock is currently owned by a thread and multiple threads are waiting for it in software. Meanwhile, an MSA entry becomes available. Then a new `LOCK` request for the same variable would allocate an MSA entry. As far as the MSA knows, the lock is free so it would be granted to this thread, thus breaking lock semantics – two threads are in the critical section, one granted entry by the software fallback and one by the MSA. The OMU prevents this situation because the counter that corresponds to the lock is non-zero as long as any thread is owning or waiting for the lock in software. When a new request is made, the non-zero counter in the OMU steers the request safely to software. Only when the thread becomes free (no thread owns it or waits for it) in software will it become eligible for MSA entry allocation when the next request is made. For high-contention locks, this may keep the lock in software for a long time. However, in all the benchmarks we used, such continuous-requests activity eventually has a "lull" in requests that allows the software activity to drain out, allowing the MSA to be used on the next burst of activity. In our evaluation we have only seen one application that shows noticeable performance degradation from this problem. In most of the cases, bursts of activity on the same lock, even when steered to software, usually "drain out" relatively quickly and allow the lock to be given an MSA entry (if one is available).

Since the OMU uses a small number of counters without tagging them, different synchronization addresses may alias to the same counter. This potentially affects performance – a synchronization variable may unnecessarily be steered to software instead of granted an

MSA entry. This can be avoided by using enough OMU counters, or even using counting Bloom filters instead of simple counters. However, the aliasing in the OMU does not affect correctness – the variable that is unnecessarily steered to software cannot have any MSA-handled operations already in progress. This is because a synchronization request always first checks the MSA. If an entry is found, the operation proceeds in hardware (no OMU lookup). The OMU lookup occurs only when the MSA entry is *not* found. Therefore, a synchronization address that already has an MSA entry will continue to use the MSA until the HWQueue becomes empty and the MSA entry is freed. This makes it possible for a synchronization variable to keep owning an MSA entry by continuously making acquire requests on that variable. Just like for software-steered streaks of requests, this is not a correctness problem and in the benchmarks we used it is also not a significant performance problem.

Note that one could eliminate OMU entirely by simply allocating/deallocating when an entry is initialized and destroyed. However, this significantly reduces the coverage of the accelerator, since from our evaluation, some applications will use more then thousands of locks. In addition, the problem becomes even more problematic when there are multiple applications. An application may end up occupying all the entries while being suspended, thus leaving active applications with no hardware resources to use.

Several proposals have opted to use a software handler solution for hardware resource overflow [14, 15]. They utilize two bits (FBIT/SBIT) to record the status of each slice of accelerator. FBIT is set/cleared when the accelerator is full/empty, and SBIT is set/cleared when there are active entries in the software. In order to provide atomicity, the software handler must acquire a per-slice lock first. In addition, the status of the accelerator needs to be re-checked because, by the time the software handler acquires the lock, the accelerator's state may have already been changed. This adds latency to each lock operation when no matching entry is found in hardware, so resource overflow needs to be very rare. Additionally, special instructions are required to let the software handler insert an entry back into

hardware, which adds complexity and requires message exchange between the core and the accelerator. In contrast, our OMU resolves resource overflow locally, and provides graceful performance degradation when resources overflow.

3.2 Synchronization Primitives

3.2.1 Lock Synchronization

Lock acquire/release is requested by a program through `LOCK`/`UNLOCK` instructions. The `LOCK` instruction results in sending a request to the MSA in the home tile of the synchronization address. If an MSA entry is already allocated for this address, or if an entry can be allocated, the HWQueue bit for the requesting core would be set to 1. If no other HWQueue bit is one, the accelerator returns a `SUCCESS` message, and the `LOCK` instruction returns `SUCCESS`, which indicates that the requesting core has acquired the lock. If the lock is currently held by another core, the HWQueue bit for that other core would be 1 and the requesting thread would not be granted the lock. In this case, the MSA simply delays the response. This prevents the requesting core's `LOCK` instruction from being committed, stalling its core until the lock is obtained.

An `UNLOCK` instruction also sends a message to the accelerator, which clears the cores' bit in the HWQueue and checks the remaining bits. If any other bit is set, one of them is selected and MSA responds to that core with a `SUCCESS` message. That core's `LOCK` instruction now returns `SUCCESS` (it acquired the lock) while the others in the HWQueue continue to wait. To ensure fairness, the MSA in each tile maintains one (for the entire MSA, not for each entry) next-bit-to-check (NBTC) register. When more than one waiting core is found in the HWQueue after an `UNLOCK`, the next core to release is selected starting at the NBTC position and the NBTC register is updated to point to the bit-position after the released one.

MSA/OMU State Diagram:

Figure 3.2 shows the state diagram of the synchronization accelerator for lock/unlock operations. Once a `LOCK` request is received, it first checks if a matching entry exists in MSA. A hit in MSA will result in handling the lock operation in hardware. A miss, however, will result in querying the OMU. If the MSA is not full and the OMU counter is zero, then a new entry is inserted into MSA and thus result in utilizing the hardware accelerator. Otherwise the OMU counter is incremented and the request is responded with a `FAIL` message.

For `UNLOCK`, if a matching MSA entry is found, then the `UNLOCK` operation is performed by MSA. Otherwise the OMU counter is decremented and the request is responded with a `FAIL` message.

Thread Suspension, Migration, and Interrupts:

When the core is interrupted for context-switching (or any other reason) while the instruction at the top of the ROB is a `LOCK` instruction, a `SUSPEND` request is sent to the lock's MSA. Upon receiving the `SUSPEND` request, the MSA clears the corresponding bit in the HWQueue, dequeueing the core from the lock's waiting list. When the thread is resumed on this (or another) core, it re-executes the `LOCK` instruction. Recall that a `LOCK` instruction that is not at the head of the ROB has not sent its request to the MSA yet, so it is simply squashed and, when the thread continues execution, re-executed.

The situation is slightly different when the thread that owns the lock is suspended. In this case, the MSA will not be notified because the `LOCK` instruction has already completed (retired). Other threads in the HWQueue continue to wait (because the lock is still held by the suspended thread). When the thread is resumed, eventually it executes an `UNLOCK` instruction that sends a message to the MSA. If the thread resumes on the same core, the MSA will behave correctly – it clears that core's bit in HWQueue and signals the next waiting core. However, if the thread resumes on another core, the `UNLOCK` request will come from a core that does not have the HWQueue bit set. In this scenario, the MSA does not know

which core originally issued the LOCK request – it is one of the cores whose HWQueue bits are 1, but we do not know which one. To resolve this situation, the MSA simply replies that the UNLOCK was successful, then replies to all cores in the HWQueue with an ABORT message, frees the MSA entry, and increments the OMU counter by the appropriate amount. This causes all waiting threads to fall back to a software lock implementation. Note that at this point the lock is free and has no threads waiting in hardware, so it is safe to fall back to software. Since our proposed mechanism has very little overhead when falling back to a software lock, this sacrifices the opportunity for hardware acceleration but does not incur a noticeable overhead beyond that.

```
   Lock(*lock) {
1  result = LOCK lock ;                                  /* execute HW lock inst */
2  if result==FAIL —— result==ABORT then
3  |   pthread_mutex_lock(lock)
4  end
5  }

6  Unlock(*lock) {
7  result = UNLOCK lock ;                               /* execute HW unlock inst */
8  if result==FAIL then
9  |   pthread_mutex_unlock(lock)
10 end
11 }
```

Algorithm 1: Modified Lock/Unlock Algorithm

Algorithm:

Algorithm 1 shows the lock algorithm adapted to use the MSA. We execute the LOCK instruction first. If this instruction succeeds, the lock was obtained in hardware and the thread proceeds into the critical section. If the LOCK instruction returns FAIL (or ABORT), we fall back to the software lock algorithm. For this fall-back, we simply use pthread_mutex_lock algorithm, but any other software lock algorithm can be substituted. The unlock operation is adapted similarly to first try to use the MSA and fall back to software if the hardware UNLOCK fails.

Interestingly, this ISA can trivially be supported by failing all LOCK/UNLOCK instructions, with little overhead (see Section 4.2) compared to code that uses the software-only

pthread implementation directly. This always-fail possibility is an important feature of our approach - it allows our ISA and synchronization library changes to be implemented without committing the processor designers to perpetual future MSA/OMU (or any other) support for synchronization.

3.2.2 Barrier Synchronization

When the `BARRIER` instruction is executed, similarly to the `LOCK` instruction, a request is sent to the MSA home tile and the corresponding HWQueue bit is set if an matching MSA entry is found. This request contains the barrier's "goal" count, which the MSA entry stores in the AuxInfo field. When the "goal" number of bits are set in the HWQueue, all those cores are sent `SUCCESS` responses. If the barrier cannot be handled in hardware[1],the accelerator immediately returns `FAIL` and the requesting core must fall back to a software implementation of barrier synchronization.

As with locks, hardware and software are prevented from simultaneously implementing the same barrier. Without this, a few arriving threads may be handled in software (e.g. because no MSA entry is available), and the rest of the arriving threads may be handled by the MSA (an entry became available). In this scenario, neither the software barrier implementation nor the MSA would ever reach the barrier's target count, which would deadlock all the threads that participate in that barrier.

MSA/OMU State Diagram:

Figure 3.3 shows the state diagram of the synchronization accelerator for the barrier operation, which is similar to the diagram for lock operation. When it receives a `BARRIER` message, the MSA checks for a matching entry. If such an entry is found, the corresponding HWQueue bit is set and, if enough HWQueue bits are set, the barrier is released (send `SUCCESS` to all participating cores). If no matching MSA entry is found, the OMU is

[1]Because no MSA entry is available, or because the OMU indicates that other threads have already arrived in software

queried and we either allocate a new MSA entry or return `FAIL`.

Thread Suspension, Migration, and Interrupts:

When a thread is interrupted while waiting at the barrier, the `BARRIER` instruction would be at the top of the ROB which results in the core sending a `SUSPEND` request to the MSA tile. However, unlike locks which will simply dequeue the requesting core, for barriers we send `FAIL` (or `ABORT`) responses to all participating cores, i.e. we force the barrier to fall back to software.

We note that it might be possible to handle thread suspend/migration in a more efficient way. An additional counter would be added to count inactive-but-arrived-to-barrier threads, and this counter would also need to be decremented when the thread resumes execution. Another source of complexity would be to ensure that all threads are correctly notified when the barrier is released – even those threads that are absent (suspended) when the last thread arrives to the barrier. This requires the hardware accelerator to keep track of which threads have been signaled and which have not yet been signalled. The approach we use (fall back to software) reduces both hardware cost and its verification complexity.

```
   Barrier(*barr) {
1  result=BARRIER barr, goal_count ;                    /* execute HW barrier inst */
2  if result==FAIL —— result==ABORT then
3  |   pthread_barrier_wait(barr) ;
4  |   FINISH barr ;                                  /* notify OMU of exiting barrier */
5  end
6  }
```

Algorithm 2: Modified Barrier Algorithm

Algorithm:

Algorithm 2 shows the barrier code adapted to use the hardware accelerator. Like for locks, the modification involves trying the hardware synchronization first and falling back to software if that fails. The only major difference is that, once the software barrier implementation exits, we send a `FINISH` request to the OMU in the barrier's home node. This

ensures the OMU to keep track of how many threads remain within the software barrier code. This FINISH instruction was not needed for locks because the exit notification was provided by the UNLOCK instruction. For barriers, a failed BARRIER instruction only indicates the entry into the software implementation, but the exit from the software barrier can be many cycles later (when all threads have arrived).

3.2.3 Condition Variable

Condition variables are supported through COND_WAIT, COND_SIGNAL, and COND_BCAST instructions. We follow standard POSIX condition variable semantics, where a wait operation waits for signals/broadcasts but also temporarily (while waiting) unlocks the associated lock.

A COND_WAIT request involves sending an UNLOCK request to the lock's home tile while enqueueing the core in the HWQueue for the condition variable. The enqueueing of a core is accomplished by setting the corresponding bit in the HWQueue. No response is sent until the core is released (by COND_SIGNAL or COND_BCAST). When no MSA entry is available for the condition variable, a FAIL response is sent back, so the COND_WAIT instruction returns FAIL, and the condition variable wait must be implemented in software.

A COND_SIGNAL instruction sends a message to the MSA home of the condition variable. If a matching MSA entry is found, SUCCESS is returned to the signaling thread, and one of the waiting cores from the HWQueue is selected for wakeup and its HWQueue bit is cleared. The next step is to re-acquire the lock that was released when that core began waiting, so we send a LOCK request to the lock's home on behalf of the waiting core. The lock home tile will then respond to the waiting core with a SUCCESS message when it eventually acquires the lock, and the COND_WAIT instruction on that core returns SUCCESS.

The COND_BCAST instruction is similar, except that it results in waking up all cores in the HWQueue, not just one. This results in multiple LOCK requests to the lock's home tile

where each has to wait to actually be granted the lock.

In our hardware condition variable implementation, the condition variable's home tile sends the LOCK request and the lock's MSA responds to the waiting core only when the lock is acquired. The associated lock address is thus stored in the AuxInfo field when receiving the COND_WAIT request. The advantage of this approach is that the COND_WAIT instruction, if successful, completes the entire condition wait operation. Another option would be to separate the condition wait into the "release lock and wait for signal/broadcast" and "re-acquire the lock we released", i.e. to have the condition variable's home respond directly to the waiting core with SUCCESS when the signal/broadcast is received, and require the lock to be re-acquired by executing a LOCK instruction. We do not use this alternative to avoid including "under the hood" workings of synchronization implementation in the ISA definition.

If no MSA entry is found for the condition variable, the home responds to the COND_SIGNAL and COND_BCAST messages with a FAIL response. When the corresponding signal/broadcast instruction completes with a FAIL result, the thread implements the signal/broadcast operation in software.

MSA/OMU State Diagram:

Figure 3.4 shows the state diagram of MSA for handling a condition wait operation. Once a COND_WAIT request is received, it first checks if a matching entry exists in MSA. A hit in the MSA will result in handling the condition variable operation in hardware, whereas a miss in will result in querying the OMU.

For OMU access, additional lock state information is used to determine the OMU response. If both the lock and the condition variable can be handled in hardware, a new entry is allocated for the condition variable. This ensures that, if a condition variable is implemented in hardware, its associated lock is also implemented in hardware. If the lock is handled in software, then condition variables that uses that lock will be handled in soft-

ware, too. This avoids the relatively complex corner case when the condition variable is handled in hardware but its lock is handled in software. The implementation of the condition wait in this case would require additional synchronization (using an auxiliary lock) to ensure correctness, which would in turn require breaking up the COND_WAIT instruction into sub-operations such as "non-blocking enqueue" and "wait for signal/broadcast" and additional complexity to handle the potential failure of each such instruction.

In Figure 3.4, the OMU indicates "HW" when it is safe to insert a new entry into the MSA. To make this decision, it needs to know whether the lock has (or can get) an MSA entry in its own home tile. Recall that for COND_WAIT requests, the condition variable's home sends an unlock request to the lock's home, and that an MSA entry for a condition variable is allocated when a COND_WAIT message is received and no MSA entry already matches it. Thus, when the condition variable's home gets a COND_WAIT request with no already-matching MSA entry, it first checks if an MSA entry is available. If not, it responds with FAIL. If an entry is available, it is reserved (but not yet allocated), and a special "unlock and pin entry" (UNLOCK&PIN) message is send to the lock's home. When it receives the UNLOCK&PIN message, the lock's home performs a normal UNLOCK attempt. If it fails, a FAIL response is sent to the condition variable's home, which frees the reserved MSA entry and returns FAIL for the COND_WAIT operation. If the UNLOCK succeeds for the UNLOCK&PIN request, the lock's home pins the lock' MSA entry so it cannot be deallocated (even if its HWQueue is empty) as long as the condition variable has an MSA entry, and then returns SUCCESS in response to the UNLOCK&PIN request. When the response is received by the condition variable's home, it changes the reserved MSA entry into an allocated one and continues with its COND_WAIT operation normally.

When the condition variable's home releases a core from its HWQueue, recall that this results in sending the lock's home a LOCK request to re-acquire the lock that was released when entering the COND_WAIT operation. If this was the last core in the HWQueue, the condition variable's MSA entry becomes free. To notify the lock's home that the condition

variable no longer requires the lock to be pinned to its MSA entry, the LOCK request sent to the lock's home in this situation is changed into a special LOCK&UNPIN request. When this request is received by the lock's home, it decrements the lock's AuxInfo counter and then processes the LOCK part of the request.

The pinning of lock MSA entries is implemented by tracking (in the lock's AuxInfo field) how many condition variables are currently "pinning" this lock. This counter is incremented when the UNLOCK&PIN request succeeds and is decremented when the LOCK&UNPIN request arrives. Note that the LOCK&UNPIN request always succeeds because, when the request arrives at the lock's home, the lock is pinned (AuxInfo is non-zero) to its MSA entry.

Thread Suspension, Migration, and Interrupts:

When a thread is interrupted while waiting at the condition variable, it returns without re-acquiring the lock. First, the core will send a SUSPEND request to the home MSA of the synchronization (condition variable) address. Upon receiving the SUSPEND request, the MSA removes the thread from its HWQueue and sends an ABORT response back. Note that this is very similar to releasing a waiting thread, except that we respond directly to the requestor without obtaining the lock. The fallback for the ABORT result of a COND_WAIT instruction is to re-acquire the lock (using Algorithm 1) and then execute a FINISH instruction. Note that the suspended/migrated/interrupted thread completes the COND_WAIT instruction and only continues to execute the fallback code when it begins to run again.

If no signal/broadcast events have actually occurred by the time the thread re-acquires the lock and exits its condition wait library call, the end result is a spurious wakeup of that thread. However, spurious wakeups of cond_wait are allowed by its POSIX semantics for very similar reasons to ours – a thread that needs to handle a signal (like SIGQUIT, SIGTERM, and other interrupt-like events, not cond_signal) needs to exit pthread_cond_wait prematurely and thus has a spurious wakeup. The spurious wakeup

possibility requires use of a while loop that re-checks the condition when the condition wait returns. If the re-check fails, recall that it still holds the lock that it re-acquired when exiting the spuriously-successful condition wait. Thus the thread can safely call the condition wait again. The essential property of this is that a condition signal/broadcast must wake up thread(s) that is waiting for it, but a thread can also be woken up even if no signal/broadcast has occurred.

Interestingly, it is possible to implement condition variables in software in such a way that eliminates the possibility of spurious wakeups. A common implementation of this approach uses timestamps to track when the last broadcast and the last "wasted" signal (no thread woken up) occurred. It is possible to use our `COND_WAIT` instruction under such semantics, but it requires the reading of these timestamps prior to attempting to do a condition wait in hardware (`COND_WAIT` instruction). When the instruction is aborted and the condition variable's associated lock is re-acquired, the timestamps would be checked again to see if we should succeed and return (signal/broadcast did occur since our wait originally began) or go back to waiting.

Algorithm:

Algorithm 3 show the modified condition_wait and condition signal/broadcast. Similar to barriers, if the condition variable is handled in software and a thread has been signalled, it also needs to send a `FINISH` message to MSA to decrement the OMU counter. Unlike locks and barriers, condition variables handle the `FAIL` and `ABORT` cases separately. As described in Section 3.2.3, an `ABORT` results in re-acquiring the lock and (possibly spuriously) returning control to the application.

We use "sw_cond_wait" as our software fallback algorithm instead of the original pthread function `pthread_cond_wait`. This is because the pthread function internally calls the software lock operations. Our `sw_cond_wait` implementation is identical to `pthread_cond_wait`, except that the lock operations it calls are the hardware-with-software-fallback lock/unlock

```
   CondWait(*cond, *lock) {
1  result = COND_WAIT cond, lock ;                          /* execute HW cond_wait inst */
2  if result==FAIL then
3  |   sw_cond_wait(cond, lock) ;
4  |   FINISH cond ;                          /* notify OMU of exiting condition variable */
5  end
6  else if result==ABORT then
7  |   LOCK(lock) ;
8  |   FINISH cond ;                          /* notify OMU of exiting condition variable */
9  end
10 }
11 CondSignal(*cond) {
12 release = COND_SIGNAL cond ;
14 if release==FAIL then
15 |   sw_cond_signal(cond)
16 end
17 }
18 CondBroadcast(*cond) {
19 release = COND_BCAST cond ;
21 if release==FAIL then
22 |   sw_cond_broadcast(cond)
23 end
24 }
```

Algorithm 3: Modified Condition Variable Algorithms

functions from Algorithm 1. This is needed because, while we prevent a condition variable from using the MSA if its lock is implemented in software, it is possible for the condition variable to be implemented in software while its lock is implemented in hardware. Therefore, the software implementation of `cond_wait` needs to use the *Lock*/*Unlock* function defined in Algorithm 1.

3.3 Optimization

Programs access the MSA through a set of synchronization instructions that send requests to the synchronization address's home tile. In the event of the operation cannot be performed in hardware, this will add an on-chip round-trip latency before it falls back to software synchronization.

For barriers and condition variables, this round-trip overhead is small compared to the overall latency of the software implementation. For locks, however, the software fallback can have low latency if the lock variable was previously owned by the same core and still resides in the core's private (e.g. L1) cache. In that case, the lock can be acquired in software without any coherence traffic. In such cases, the added round-trip latency to consult

the home's MSA/OMU adds an overhead that is not negligible relative to the latency of the software fallback alone. A potential optimization would be to profile the application and identify locks that are both low-contention and acquired quickly (L1 hit), and not attempt to use the hardware for such locks at all. However, we prefer solutions that avoids placing additional burdens on application programmers (our modified synchronization algorithms only require changes to the synchronization part of the runtime library). Therefore, we propose an optimization that allows skipping many doomed-to-succeed MSA/OMU checks transparently to both synchronization library and application programmers.

The optimization uses the presence of a (writable) cache block that contains the synchronization address as an proxy for "can acquire the lock without informing the home". When the hardware accelerator grants the lock ownership to a core, along with replying the request with a "SUCCESS" message, it also grants the core an exclusive ownership (E state in the MESI protocol) of the cache block, invalidating any other cached instances of this block.

Upon receiving the cache block and (successfully) completing the `LOCK` instruction, the core will put the block in its L1 cache and set the "HWSync" bit (a new bit that is added to each line in the cache) for its cache line. This bit indicates that the core was the last one to successfully complete a hardware lock operation for that cache line. In contrast, a normal read or write request will bring in the cache block without setting the "HWSync" bit. Note that, since the synchronization accelerator resides with the home node of the cache block, it can easily retrieve the cache state information of a particular cache block and cause the block to be sent along with the response.

The `UNLOCK` instruction does not clear the HWSync bit. When that core issues the next `LOCK` request, if its L1 cache still has the cache block with "HWSync" equal to one, the core can send a `LOCK_SILENT` notification to the home tile of the lock but its `LOCK` instruction can return "SUCCESS" immediately. This notifies the MSA that the core has re-acquired the lock, allowing the MSA entry to be updated, but avoids adding the round-

trip latency if the lock is quickly re-acquired by the same thread that held it previously.

3.4 Evaluation

We evaluate synchronization approaches using SESC [63], a cycle-accurate architectural simulator. We model 16-core and 64-core processors, with 2-issue out-of-order cores and private IL1 and DL1 caches. The L2 cache is a distributed shared last-level cache, so each core has a slice of the L2 cache and a router for the packet-switched 2D mesh network-on-chip (NoC). We model the NoC using Booksim [64], a cycle-accurate NoC simulator that we integrated into SESC.

In our evaluation MSA/OMU-N models our hardware synchronization with an N-entry MSA and a four-counter OMU in each slice. We also evaluate MSA-0 configuration, which does not have any hardware synchronization support and trivially implements our instructions by always returning `FAIL` (without sending a message to the home node). This configuration is used with the same modified synchronization library, so it shows how much overhead would be added by these modified algorithms in a machine that does not provide actual MSA/OMU hardware, e.g. if the new instructions are adopted to exploit our MSA/OMU hardware and then this hardware is eliminated in some future versions of the processor. Another configuration we evaluate is MSA-inf where we model a MSA with an infinite number of entries (so no OMU is needed). This configuration provides insight into how much performance is lost due to limited MSA size.

The benchmarks we use are the (entire) Splash2 [28] and PARSEC [65] benchmark suites. All benchmarks are complied with the GCC 4.6.3 compiler suite using -O3 optimization. For non-baseline runs, we replace the pthread synchronization library calls with more advanced software implementations (MCS lock and tournament barrier [66]), synchronization library that utilizes algorithms discussed in Section 3.2, along with different types of MSA (MSA-$0/inf$, or MSA/OMU), depending on the synchronization approach used in that run.

3.4.1 Raw Synchronization Latency

Figure 3.5 shows the raw cycle count directly attributable to synchronization, excluding the waiting time that would be present even with an ideal (zero-latency) synchronization. Note that this figure uses a logarithmic scale.

We model the no-contention case for locks using disjoint sets of locks in different threads, and measure the time between entering and exiting the `lock()` function. All synchronization approaches perform similarly in this case, except for MSA/OMU-2; because for no-contention, our HWSync-bit optimization scheme results in most `LOCK` instructions to succeed without waiting for the MSA's response. This avoids both the overheads of software implementations and the round-trip latency of a non-optimized hardware implementation. The high-contention case is modeled by having all threads access the same lock. Lock handoff is measured from the cycle in which a thread enters `unlock()` to the cycle in which the released `lock()` exits. In this case, pthread_mutexlock and spinlock have high handoff latency with a poor scaling trend (from 16 to 64 cores). The more scalable MCS lock has a faster handoff and scales better than the pthreads lock implementations. With high contention, our MSA/OMU-2 configuration does not benefit from the HWSync-bit optimization, but nevertheless has the lowest handoff latency and best scaling trend because the MSA implements lock handoff efficiently.

For barriers we measure latency from the time that the last-arriving thread enters `barrier()` to the time all threads have exit. Our MSA/OMU approach provides an order-of-magnitude improvement over the best software implementation (tournament barrier).

For condition variables, the latency is measured from entering `cond_signal()` or `cond_broadcast()` to the exit from the released `cond_wait()`. The MSA/OMU-2 configuration improves significantly over the software-only implementation. Part of the reason for this improvement is from improving the latency of condition variable notifications, but another reason for the improvement is that MSA/OMU-2 also provides quick handoff of the lock associated with the condition variable.

In all these cases, *MSA-0* incurs a minimum overhead compared to the baseline (pthread) scheme. This shows that our modifications to the synchronization library do not result in significant overheads when using the fallback path, i.e. if a processor does not have hardware support, it can trivially implement our ISA extensions and use the same hardware-capable synchronization code. This may be an important consideration for processor manufacturers - after adding the synchronization instructions and our MSA/OMU hardware, the processor manufacturer can drop MSA/OMU support in future generations of the processor without breaking compatibility with software that uses the new instructions.

3.4.2 Benchmark Evaluation

Figure 3.6 shows the overall application speedup, relative to the pthread baseline, for Splash and PARSEC. The averages shown are for *all 26 benchmarks in Splash and PARSEC suites*, but to reduce clutter we show in the figures only those individual applications where *Ideal* synchronization shows at least 4% benefit compared to the baseline.

The *MSA-*inf results are on average within 3% of the Ideal (zero-latency) case. Where differences are noticeable, they mostly come from message latencies to and from the synchronization variable's home. The difference is largest in 64-core execution of *radiosity*, where lock synchronization is frequent, but with many low-contention locks. Furthermore, each lock tends to be used by different threads, so our HWSyns-bit optimization hides the round-trip communication latency for only 20% of lock acquire requests. For *fluidanimate*, the difference between MSA-inf and *Ideal* is 8%. This application also has frequent operations on low-contention locks, but each lock tends to be used by the same core, allowing our HWSync-bit optimization to hide round-trip communication for 90% of lock requests.

Interestingly, *Ideal* synchronization in *ocean-nc* with 16 threads performs worse than the software baseline. We verified that the time spent on synchronization is dramatically improved (only the necessary waiting time remains) in *Ideal*, but the non-synchronization code executes with a lower IPC, primarily due to increased burstiness of cache misses (all threads leave the barrier in the exact same cycle). This "better is worse" effect is also present (to a lesser degree) in other synchronization-accelerated configurations.

Among realistic hardware implementations, *MSA/OMU-1* configuration achieves average performance within 6% of the *MSA-*inf, and *MSA/OMU-2* performs similar to *MSA-*inf. We conclude that, with the OMU, few MSA entries are needed to achieve most of the hardware-synchronization performance potential.

The *MSA-0* results are within 1% of the baseline software implementation. This confirms that our synchronization library and the ISA modifications can be implemented across entire processor families, even if some processors in those families have no actual MSA/OMU hardware. Another interesting point is for *radiosity* and *raytrace*, *MSA-0* actually shows speedup compared to the baseline. For *radiosity*, the speedup comes from the reduction of empty task queue searches, which results in 47% decrease of lock accesses. For *raytrace*, the amount of lock access did not show any signification changes. However, the average lock handoff latency for the most-contented lock was reduced by 2X. This difference comes from the changes in lock acquire order, which would affect the lock handoff latency under our distributed shared last-level cache with non-uniform cache-to-cache transfer latency.

Finally, *MCS-Tour* benefits applications with high-contention locks or frequent barrier operations. For *fluidanimate*, *MCS-Tour* shows some performance loss because MCS locks have larger overhead for no-contention locks. On average, *MCS-Tour* shows a 24% speedup, but *MSA/OMU* achieves an additional 19% speedup over this advanced software implementation.

3.4.3 Coverage Improvement from OMU

Figure 3.7 shows the percentage of synchronization operations, averaged across all Splash-2 and PARSEC benchmarks, handled by the MSA with and without the OMU. Without the OMU, MSA entries cannot be safely deallocated, so the very first synchronization variables that are used by the application are the ones that get MSA entries (and keep them "forever"). We observe a significant increase in coverage of synchronization operations with the OMU. For 64-tile MSA-2, for example, the OMU allows 93% of operations to utilize the MSA, compared to only 56% without the OMU. More importantly, the OMU naturally handles the transition from using one set of variables to another, e.g. when one application ends and another begins. Without the OMU, a separate mechanism would be needed to inform the MSA when the synchronization variable address that allocated an entry is no longer used for synchronization.

3.4.4 Lock Optimization

Figure 3.8 shows the speedup achieved in *fluidanimate* with and without the HWSync-bit optimization. Recall that the optimization allows a core to acquire a lock which it previously held (if its block is still in the L1 cache) without waiting for the lock home's response. *fluidanimate* uses many locks, but has low lock contention because each lock tends to be acquired by the same thread repeatedly. Without the HWSync-bit optimization, it is often the case that the software lock (that hits in the L1 cache) has lower latency than the hardware one (request to MSA, wait for response). This increased latency cancels out the gains which is provided by MSA/OMU, which leads to a slowdown in a 64-core machine. With the HWSync-bit optimization, the hardware locks are uniformly lower-latency than software implementation, so MSA/OMU performance shows a speedup versus a software implementation, and this speedup increases with the number of cores.

3.4.5 Synchronization Breakdown

Figure 3.9 shows the speedup for supporting only one type of synchronization (locks *or* barriers) by the MSA in a 64-core machine. For barrier-intensive applications such as *ocean/-nc* and *streamcluster*, the speedup is lost when MSA only supports locks. For lock-intensive applications, such as *radiosity* and *fluidanimate*, most or all of the speedup is lost when only supporting barriers. Interestingly, *raytrace* is a lock-intensive application, but it shows a lower speedup for *MSA-LockOnly* than for *MSA-BarrierOnly*. This is because, in *MSA-LockOnly*, the absence of barrier handling results in different allocation of MSA entries, causing one of the more contented locks to suffer more software fallback. However, the speedup becomes similar to *MSA/OMU* when we increase the MSA entries from 2 to 4.

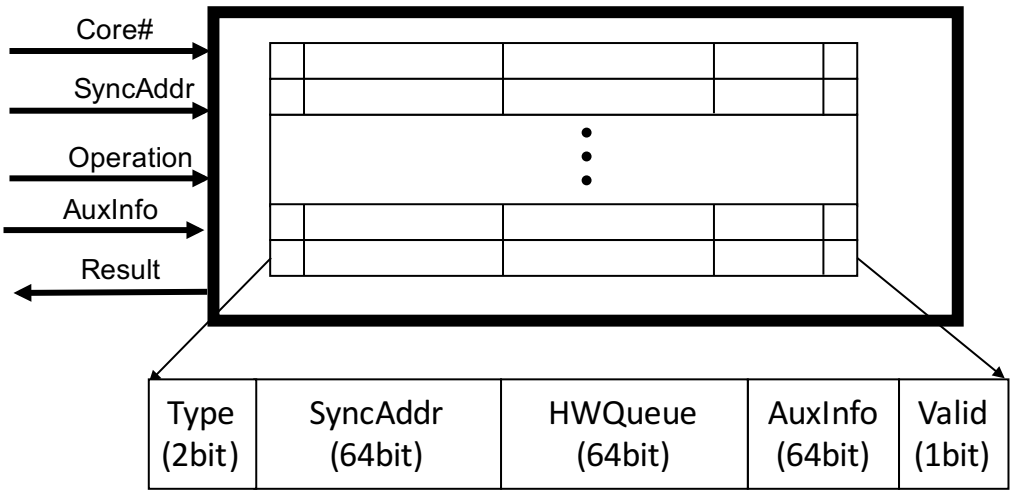

Figure 3.1: Minimalistic Synchronization Accelerator (MSA)

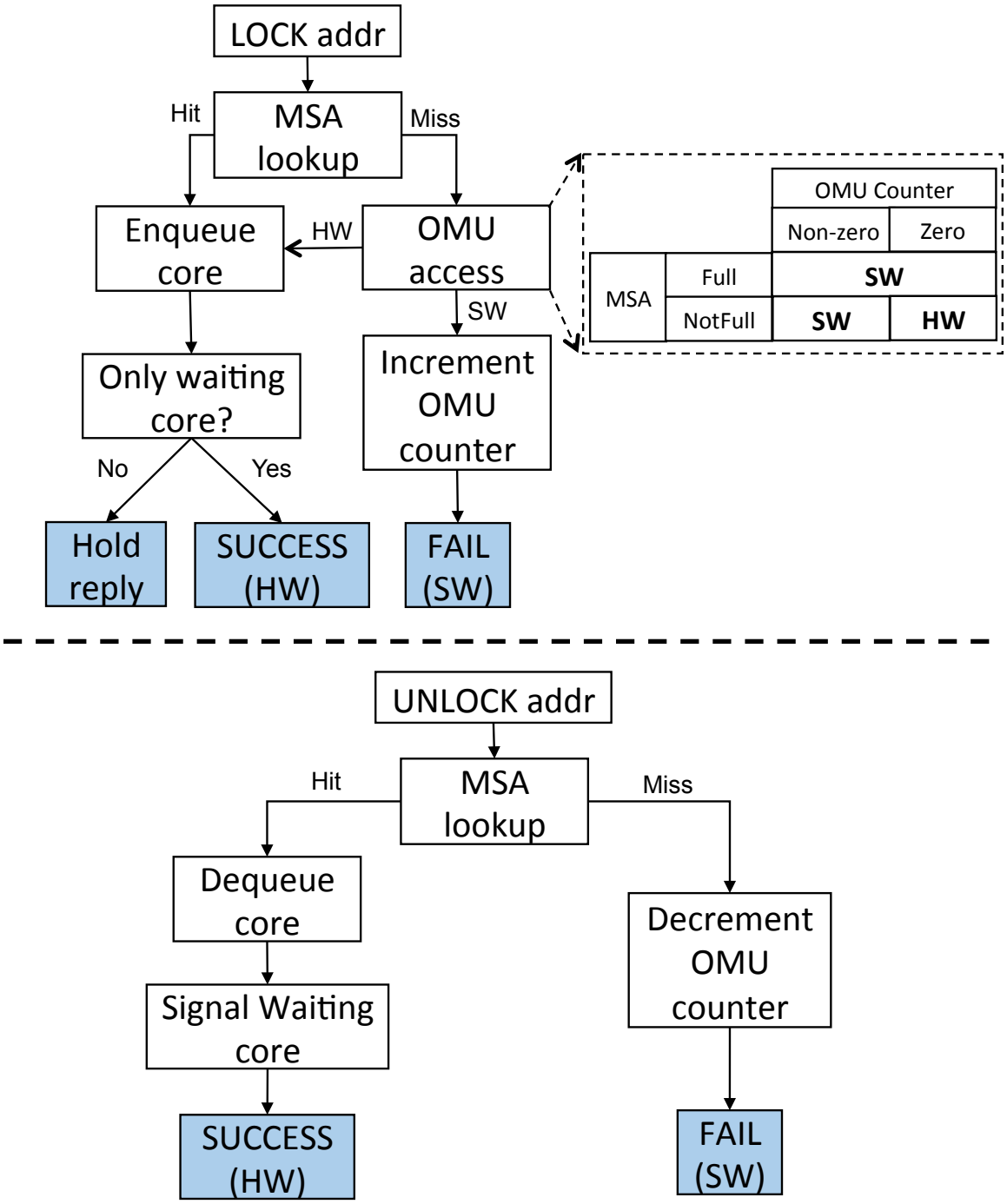

Figure 3.2: State Diagram for Lock/Unlock Operations

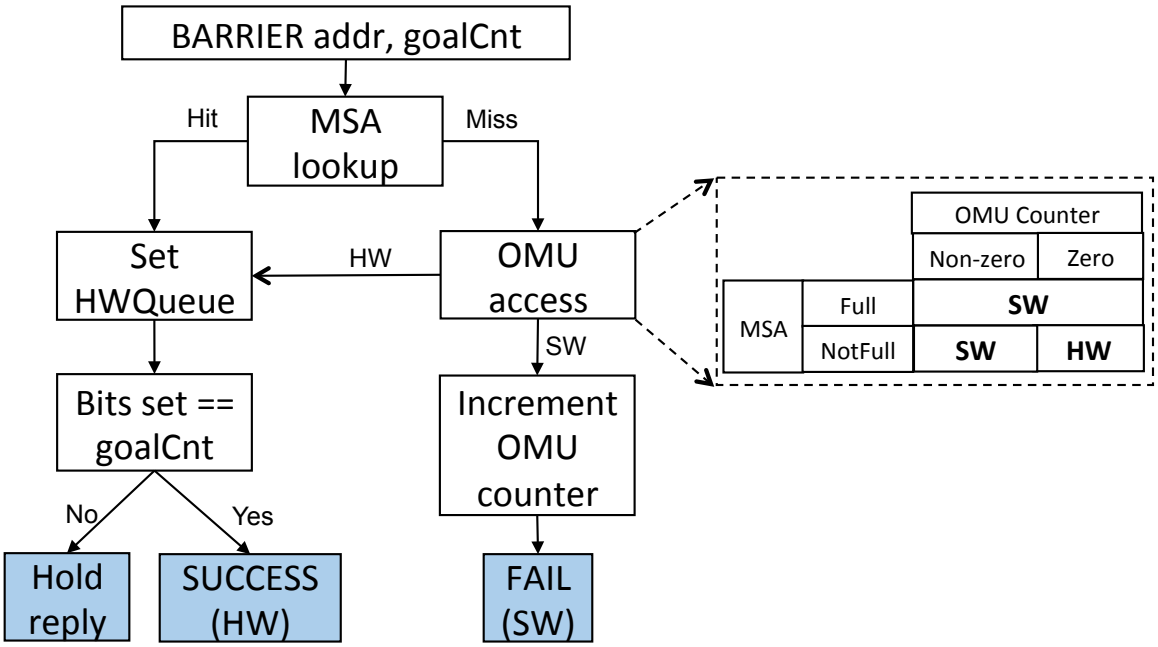

Figure 3.3: State Diagram for the Barrier Operation

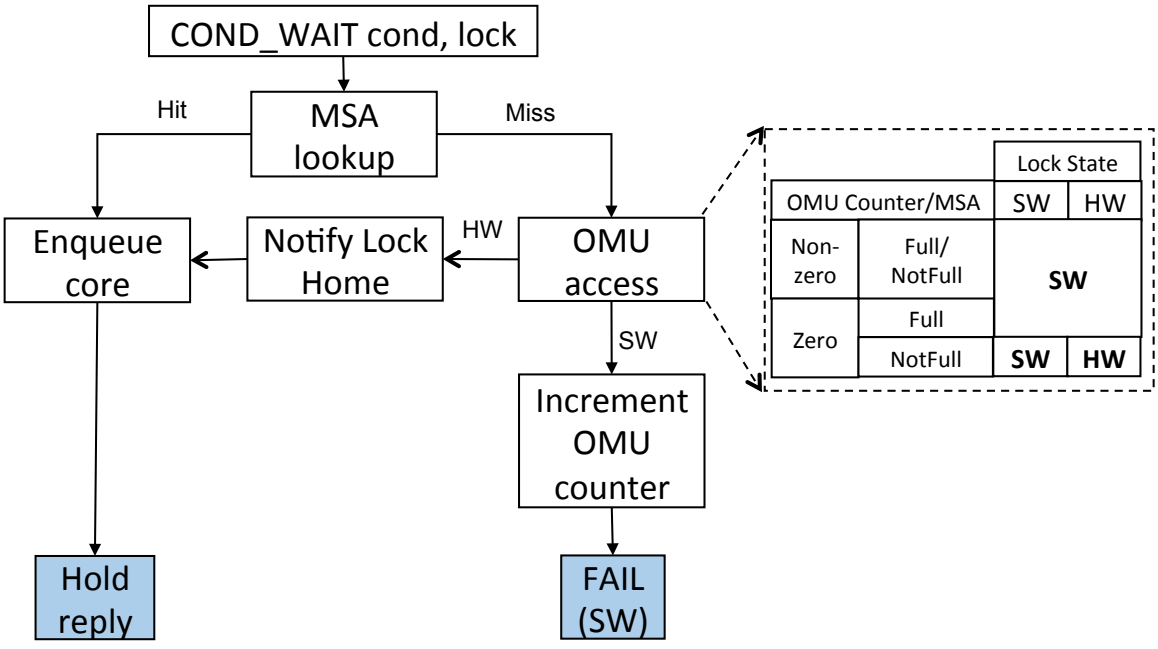

Figure 3.4: State Diagram for the Condition Wait Operation

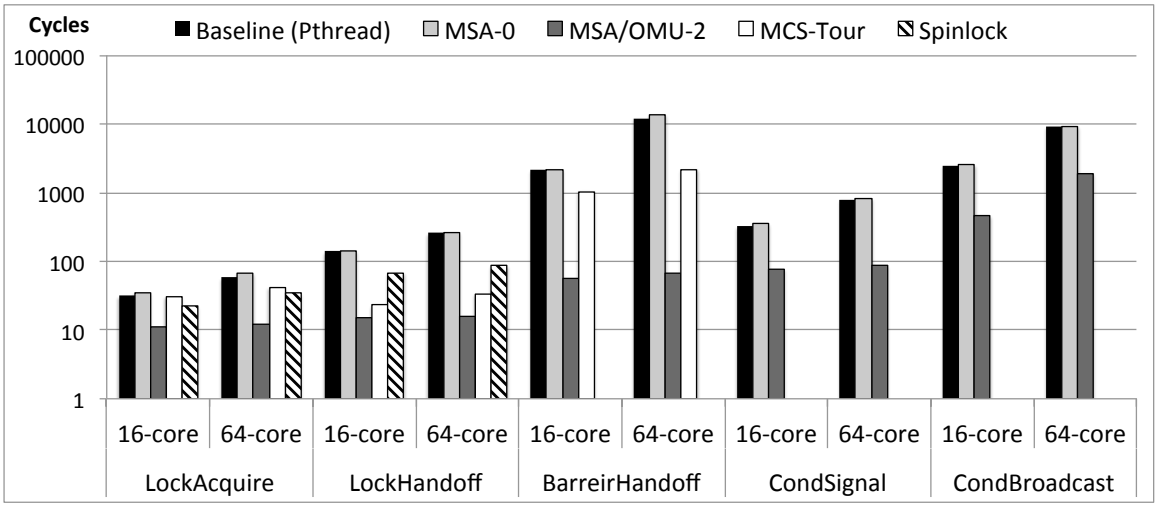

Figure 3.5: Raw Synchronization Latency

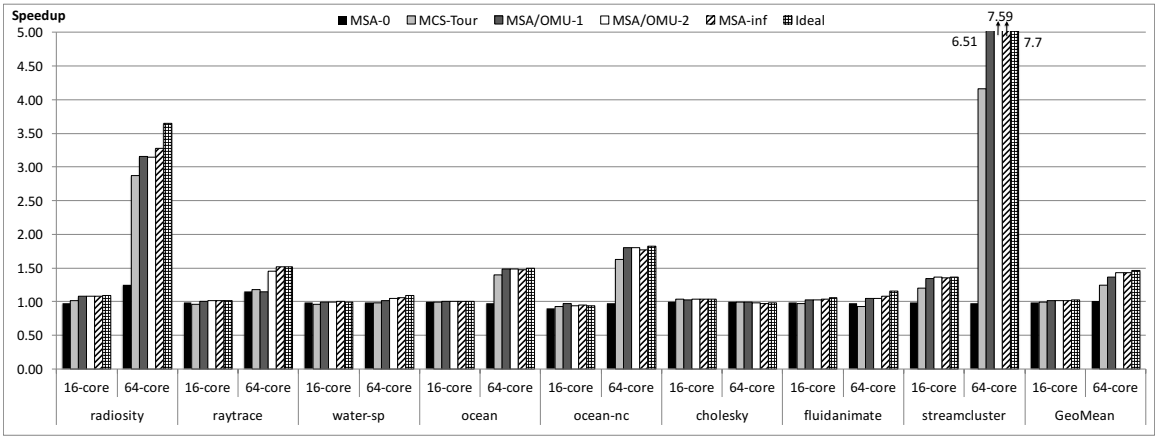

Figure 3.6: Overall application performance improvement (Speedup)

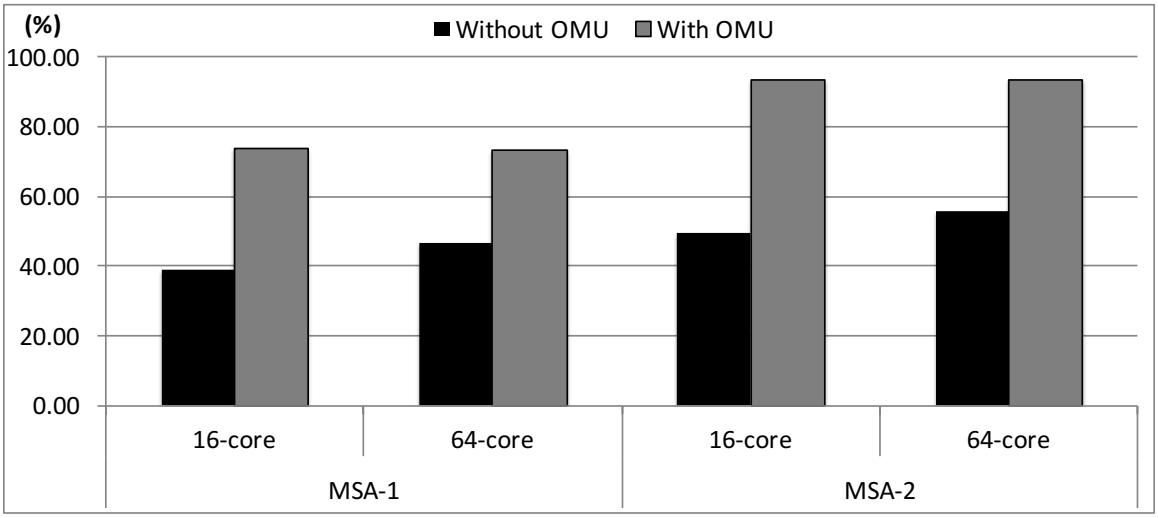

Figure 3.7: Coverage of Synchronization Operations

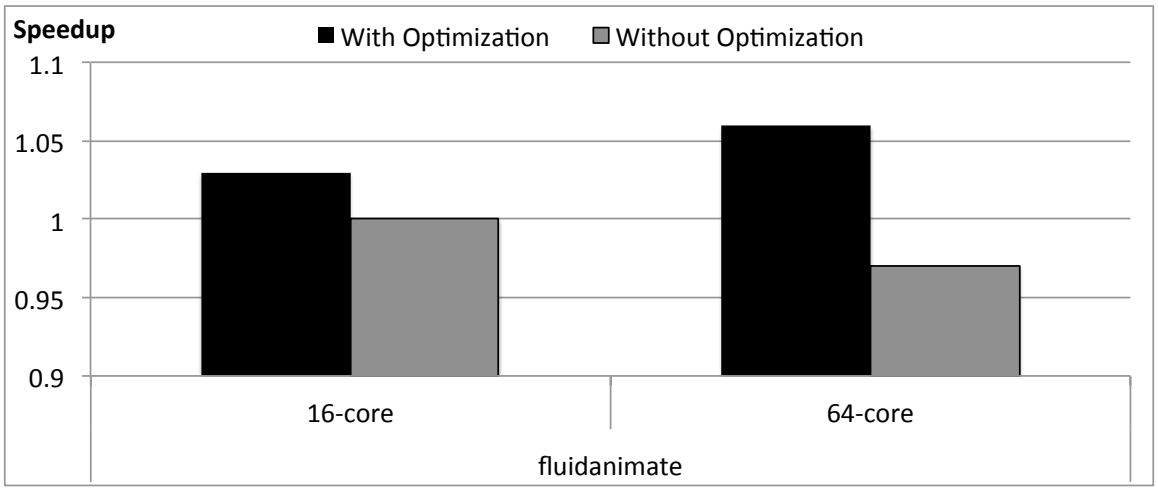

Figure 3.8: Effect of HWSync-bit optimization on Fluidanimate

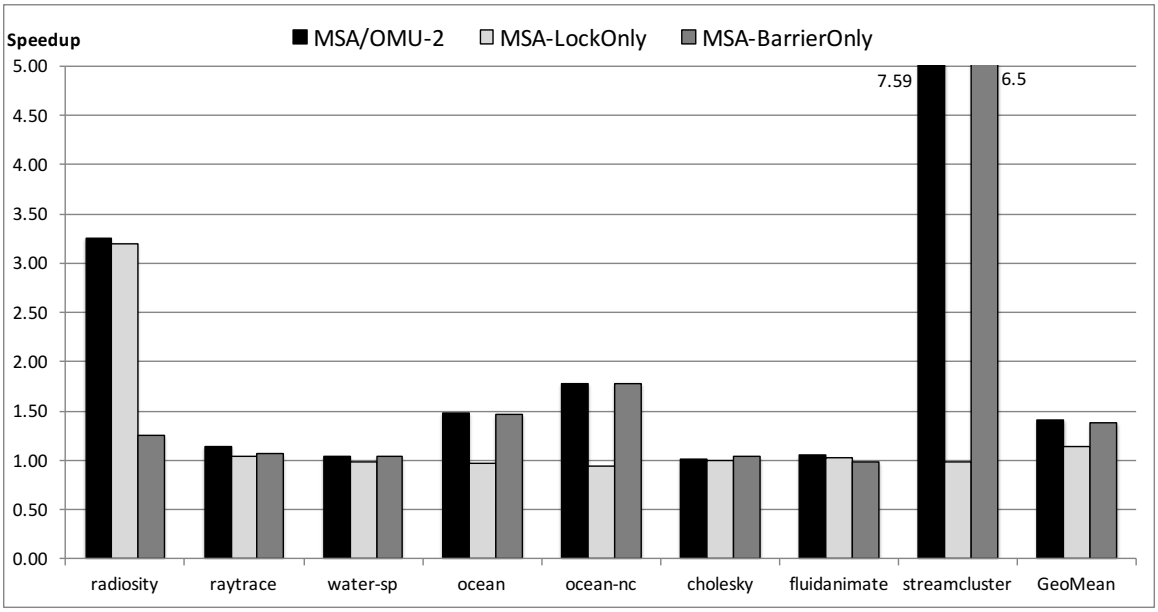

Figure 3.9: Speedup comparison when MSA only supports lock or barrier operation

CHAPTER 4

PARALLEL SPEEDUP PREDICTUON FOR MULTI-THREADED APPLICATION VIA STATISTICAL MODELING OF PROGRAM CHARATERISTICS

In this chapter, I present a new performance model that captures program characteristics of multi-threaded applications, allowing it to use few-threaded runs along with small input sets to predict performance of many-threaded runs with large input sets. The model classifies the application scaling into memory-bound or compute-bound, by predicting how the cache miss rate would change and how the DRAM memory subsystem would react to the change in memory bandwidth requirement.

4.1 Model Structure

To account for different parts of the application having different per-thread instruction count overheads, different cache behaviors, and different memory bandwidth demand, we partition the program execution into barrier phases and model performance scaling of each barrier phase. Subsequently each barrier phase is then subdivided into smaller intervals in order to capture the burstiness of program phases. This is illustrated in Figure 4.1, where an actual barrier phase execution with N threads is divided into M intervals, where the i-th interval lasts X_i cycles, has $Y_{i,total}$ instructions, and a ratio of memory instructions to all instructions is $m_i = Y_{i,memory}/Y_{i,total}$ (we call this the memory instruction ratio). Each interval represents $100/M\%$ of the total instructions in the barrier phase. Our method predicts how each interval's execution time scales when increasing the number of threads to some larger number N', then puts the intervals back together to predict the barrier phase execution time with N' threads. This allows the parallel speedup for N'-thread execution to be computed and, by computing the speedup for various desired values of N', to predict the overall performance scaling trend for that application. Finally, we model the barrier

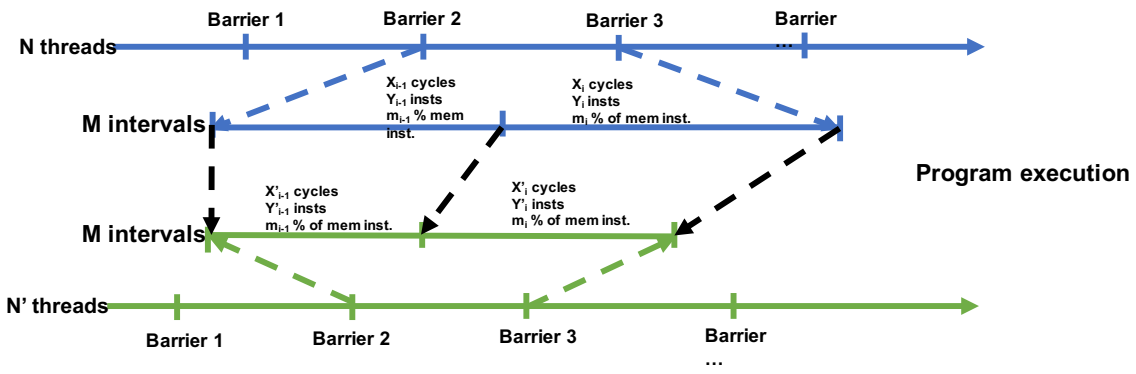

Figure 4.1: Performance Modeling for Program Intervals

phase change across different inputs and predict how barrier phase scales across different input sets.

We first classify each interval as either compute-bound or memory-bound. First, we model how the overall pro- gram characteristic changes when increasing the number of threads. Two important features in this step are the number of all instructions and the number of memory instructions. This allows us to model how the total work changes and how work is distributed with increasing number of threads. Second, we model how memory locality changes with increased thread count. This captures how the per-thread number of memory requests changes with the number of threads. Then, we evaluate how the memory subsystem affects the overall performance scaling. Last, we model how the barrier phase program characteristics change across input sets. Note that a possible improvement would be to extend interval classification to include other types of intervals, e.g. lock-synchronization-bound intervals. In our evaluation, we do see applications that would benefit from including lock-synchronization-bound classification, and is one possible improvement for future.

4.1.1 Instruction count

Instruction count is the first-order approximation on how the application's overall amount of work scales with increasing number of threads. Ideally, as the thread count increases

while the input remains unchanged (strong scaling), the same total amount of work is equally divided among the threads. This implies that the total amount of work is constant as the thread-count changes, and this has been the assumption made in prior works on performance prediction in parallel applications. This assumption holds relatively well for some applications. For example, in *FFT* and *Barnes* from the Splash-2 suite, when scaling from 2 threads to 256 threads the change (increase) in total instruction count is $<10\%$. However, this assumption is substantially "broken" for other applications, for example in *Cholesky*, where the 2-to-256 thread-count scaling causes the total instruction count to increase by 17X. This is because a non-negligible part of the overall work must be performed by each thread. As the thread count increases, each thread gets the same amount of per-thread work along with a ever-smaller portion of the work that can be divided among threads. This is mostly equivalent to having a serial section in the application, and the first-order effect of having a constant per-thread section of the work can be captured by applying Amdahl's law to this scenario. One important difference between this and the traditional serial-section scenario, however, is that in the serial-section scenario one thread is active while the others are waiting (idle) so the execution time of the serial section is largely independent of the thread count, while in the constant-per-thread-work scenario the threads are all active, consuming power, and producing cache capacity and memory bandwidth demand. Thus, the increase in instruction count, when present to a significant degree, can dramatically affect the overall performance of the application and is very important to model.

Prior work also makes the assumption that work is evenly distributed among threads. Although this can greatly simplify the overall performance model, it removes the effect of load imbalance on overall performance. Unfortunately, in some applications this assumption deteriorates as the thread count increases. Figure 4.2 shows the ratio between per-thread-maximum and per-thread-average instruction counts, as well as total instruction counts for *Cholesky* and *Blackscholes*. For *Cholesky*, in addition to an 18-fold increase in total instructions, we observe an increase in the maximum-to-average ratio (by a factor

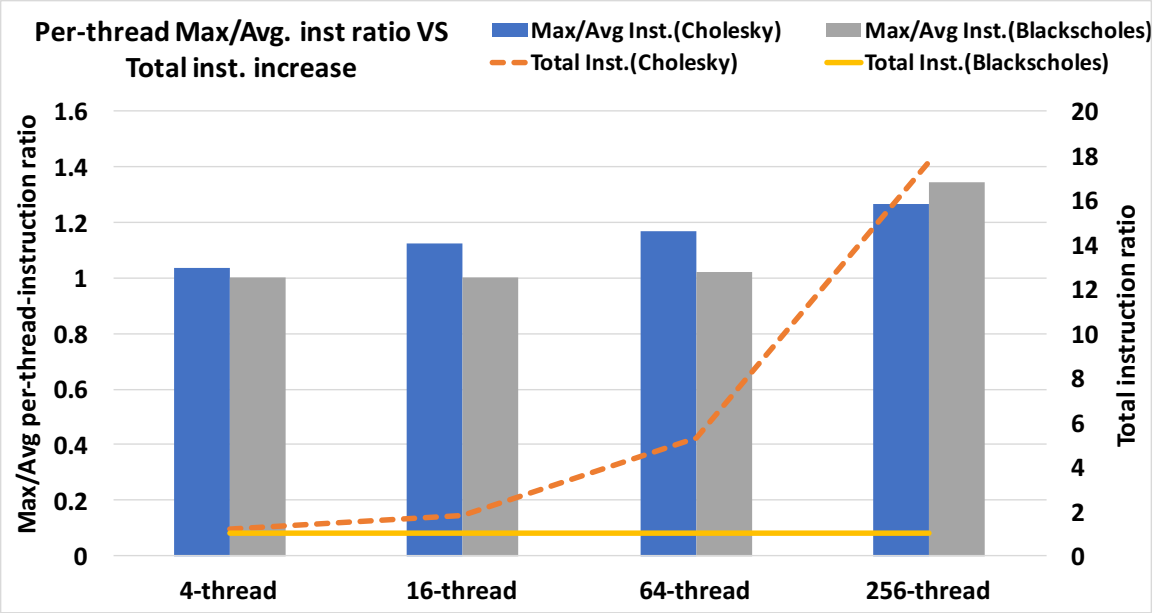

Figure 4.2: Per-Thread Max/Avg Instruction Ratio

of 1.26X). This shows that, in addition to having to do more work at higher core counts, this work is also less evenly distributed, i.e. as the core count increases the fraction of time that will be spent at a barrier, thread-join, or other synchronization will increase. For *Blackscholes*, the total instruction remains constant, but the maximum-to-average ratio increases from nearly 1X (almost-perfect work balance among threads) to 1.35X. To capture the effect of deteriorating work balance among threads, we include the dispersion of instructions(work) among threads in our model.

To model the total work and the distribution of this work among threads, we first gather the per-thread instruction count profile for several low-thread-count executions (in our evaluation we use 1-, 2-, 4-, 8-, and 16-threaded executions). The instruction count can be collected using hardware performance counters, profiling tools such as PIN [67], or a functional simulator which can efficiently record the number of dynamic instructions. In our evaluation, we use the frontend simulator[1] of the SESC [63] simulator to record the number of instructions executed by each thread. We model the instruction scaling as either a linear

[1]The front-end simulator is a functional (ISA-level) simulator, i.e. it only models the effect of the instruction on the architectural state, but not the timing and microarchitectural state, of the system

model ($y_n = a/n+b$) or a power model ($y_n = a*n^b$) and select the model with the highteset R-squared value. Here, y_n is the instruction count for a thread in n-threaded execution and a and b are parameters to be obtained through regression. For linear model, a corresponds to the amount of work that can be equally divided among threads, and b corresponds to the amount of per-thread work; for power model, a corresponds to the amount of work that can be equally divided among threads, and b corresponds to the degree of distribution of per-thread work. To account for load imbalance, instead of modeling the average instruction count among threads, we model the maximum instruction count among threads, i.e. in each execution we select the highest-instruction-count thread, use regression to fit a and b to that.

In order to further model the impact of the memory subsystem, one might argue to model the change of memory instructions, or more specifically, the trend of load and store instructions. In our study, we found that the percentage of memory instruction remains stable across different thread count, which lead us to only predict the per-thread-instruction change and assume the per-thread memory instruction follows the same trend. This simplifies the model such that $m_i = m_j$, for $i \neq j$.

4.1.2 Memory request

RD analysis is a powerful tool to analyze how the cache size affects the overall memory accesses. By recording the number of distinct cache lines accessed between two accesses to the same cache line, one can construct a reuse distance histogram and calculate the number of cache misses for a given cache size (assume LRU replacement policy). However, with parallel applications, the interleaving of memory instructions from different threads will change the temporal locality while scaling the number of threads [26].

Two main effects occur when memory streams are interleaved, *dilation* and *intercept*. When two threads are accessing private data, the interleaved memory streams will dilate each other, resulting in an increase in the reuse distance. This is known as *dilation*. On

the other hand, when two threads access the same cache line, they will shorten each other's reuse distance for that line, which is called an *intercept*.

Prior works have studied how to apply reuse distance analysis for multithreaded application, including constructing the concurrent reuse distance (CRD) profile from each thread's local reuse distance profile [30], or predicting the profile change of CRD [26]. M.J. Wu et al. [26] noticed that for loop-based multithreaded applications, the CRD profiles either shifts to larger RD or spread out with thread count scaling. Hence, they propose to utilize reference groups in order to predict the CRD profile change.

We adopt a similar approach by M.J Wu [26] to predict the number of memory requests, which utilizes the shifting and spreading property of CRD when scaling the number of threads. By constructing the accumulated CRD profile and predicting the shifting behavior for each percentage of memory access, we can predict how the cache miss rate increases with thread count scaling. We further extend the work of M.J Wu [26] to include cold miss predictions (∞-reuse distance), and we combine it with the instruction count and memory bandwidth modeling to predict the overall parallel performance rather than only the number of memory requests.

To summarize the approach for spreading and shifting the CRD profiles, Figure 4.3 shows the actual accumulated CRD profile normalized to the total memory access for *Barnes*. As shown, the CRD profiles shifts with increasing number of threads, this is the behavior of *dialtion*, which increases the reuse distance when memory accesses from different threads are interleaved. Hence, we can learn the shifting amount ($\delta_{1,2,3,4}$) from the training runs shown as solid lines (1-16 threads), and then predict the CRD when the number of threads increases (dotted lines). This is performed for each sampled interval and for each percentage of access. For simplicity, we assume the shifting amount (δ_N) follows a linear model. By using CRD profile prediction, we can predict how the cache miss rate changes when scaling the number of threads. This is denoted as $CMR(\$_{Size}, N'_{thread})$. Together with instruction count prediction, we can predict the overall number of memory

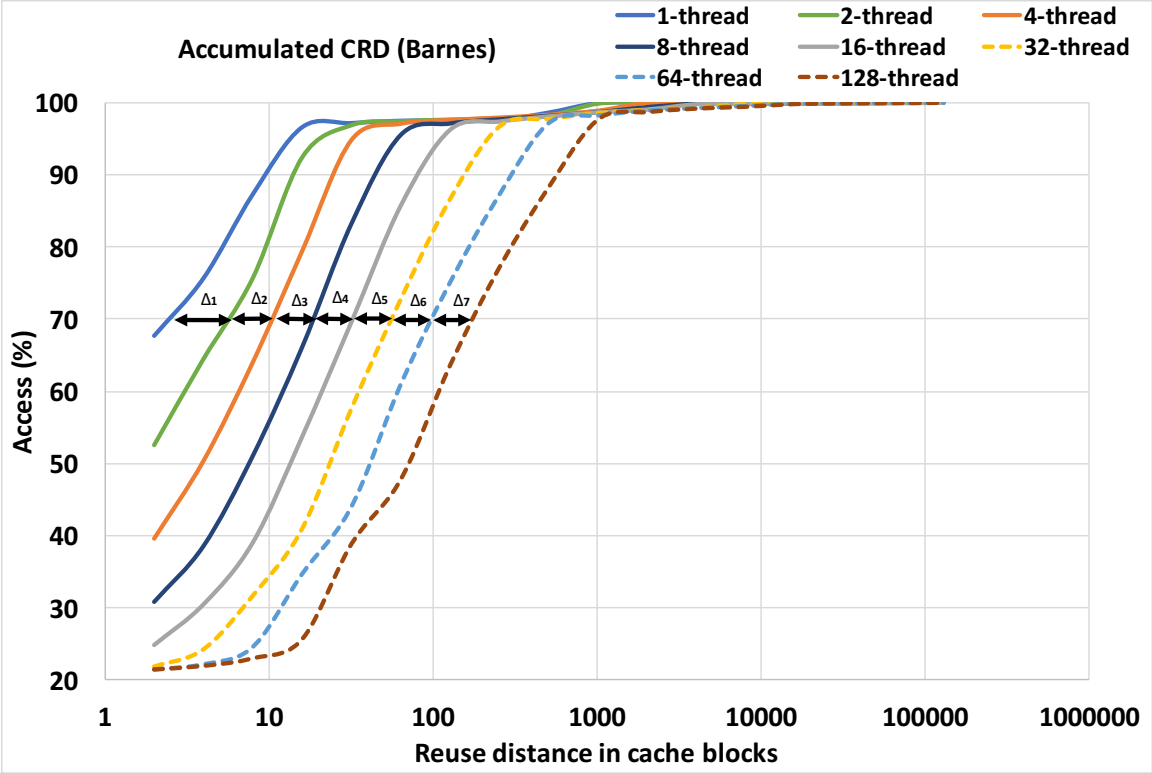

Figure 4.3: Accumulated concurrent reuse distance profile for *Barnes*

accesses under thread scaling.

Equation 4.1 shows the equation for predicting the number of memory request $N'_{i,memory}$. First, the instruction count for interval-i (Y'_i) is calculated using the per-thread-average instruction model discussed in Section. 4.1.1 and multiplying it by the number of thread N'(Eq. 4.1a). Then, assuming the memory instruction mix m_i does not change under thread scaling, we can calculate the number of memory accesses using the cache miss rate ($CMR(\$_{Size}, N'_{thread})$) from CRD profile prediction (Eq. 4.1b)

$$Y'_i = (a_{avg}/N' + b_{avg}) * N' \tag{4.1a}$$

$$N'_{i,memory} = Y'_i * m_i * CMR(\$_{Size}, N'_{thread}) \tag{4.1b}$$

Note that CRD profile only captures the effect of cold and capacity miss, which ignores the additional conflict miss traffic to the memory. For conflict misses, J.S. Harper [68] pro-

posed an analytical model for predicting cache misses on set-associative caches. However, in our evaluation the LLC has sufficient set associativity to make conflict misses a very small fraction of all misses regardless of thread-count, so we avoid over-parametrizing the overall model by omitting the modeling of cache associativity.

4.1.3 DRAM latency

CRD profile prediction allows us to obtain the number of memory accesses for each sampled interval. Therefore, the next step is to evaluate how the memory subsystem consumes the burst of memory request. Various DRAM model has been proposed in the past to explore the design space of the memory subsystem. Methods such as using an M/D/1 queueing model [57] to estimate memory latency or probability model [56] to predict bandwidth utilization. These models use statistical parameters which requires a stable and long execution to collect such information. However, since we sample and predict the program in short intervals, execution behavior would not achieve a stable condition for statistical modeling.

Instead, we propose a simple linear DRAM service time model to estimate the average service latency within a given interval. Figure 4.4 shows the average memory queue length verses average service time for *Barnes* and *Cholesky*. For short memory bursts, since not enough memory level parallelism (MLP) is available for overlapping memory requests, the service rate is simply the average time of row buffer hit and row buffer miss latency. As the memory burst length increases, more MLP results in shorter average service time and in result achieves the maximum throughput. Hence, the DRAM service time can be approximated using the average queue length to determine the average service time.

The overall service time can be calculated as follow:

$$T(N_q) = \begin{cases} T_{max} = \dfrac{T_{hit} + T_{miss}}{2}, & \text{if } N_q < N_{min}. \\ T_{min} = T_{burst} & \text{if } N_q > N_{max}. \\ a * N_q + b, & \text{otherwise.} \end{cases} \tag{4.2a}$$

$$a = \frac{T_{min} - T_{max}}{N_{max} - N_{min}} \tag{4.2b}$$

$$b = \frac{(N_{max} - N_{min} - 1) * T_{min} + T_{max}}{N_{max} - N_{min}} \tag{4.2c}$$

, where T_{hit} and T_{miss} represents the row buffer hit/miss latency and T_{burst} is the memory bus bandwidth. N_{min} and N_{max} represents the queue length threshold, which in our evaluation are 0.5 and 3.5, respectively. Note that here we assume that memory requests are evenly divided memory channels, and that the effect of bank level parallelism (BLP) and row-buffer-hit ratio is reflected in N_{min} and N_{max}. As with other potential refinements of the model, we chose whether to refine the model based on whether there was sufficient evidence that the lack of model refinement is causing systematic errors in several applications, and concluded that modeling of the imbalance among memory channels and detailed BLP modeling had little impact on our results and thus choose to simplify the model by not including more detailed models of these memory-system considerations.

4.1.4 Cycle-Count Prediction for an Interval

The overall prediction of the number of execution cycles needed for an interval is achieved as follows. First, we compute the expected cycle-count assuming the interval is compute-bound, i.e. that the IPC of each thread is constant under thread scaling. Thus the compute-bound cycle-count $X'_{i,compute}$ scales linearly with instruction count scaling and can be estimated as in Equation 4.3. Eq. 4.3a predicts the per-thread-maximum instruction using the regression model discussed in Section. 4.1.1. Together with the assumption that CPI_i

remains identicaly, we can predict the cycle-count $X'_{i,compute}$ as in Eq. 4.3b.

$$Y'_{i,max} = a_{max}/N' + b_{max} \tag{4.3a}$$

$$X'_{i,compute} = Y'_{i,max} * CPI_i \tag{4.3b}$$

Second, we predict the expected cycle-count assuming the interval is memory bound. We first estimate the number of memory accesses by estimating 1) the number of memory instructions and 2) the change in cache miss rate due to CRD scaling and the increase in memory footprint (cold misses). We then combine this with the estimated number of cycles from instruction-count scaling to calculate the expected memory queue N_q using Liitle's Law, assuming the worst-case DRAM service time T_{max}. Equation 4.4 shows the equations for predicting the expected cycle-count for a memory-bound interval. First, we use the the number of memory accesses, $N'_{i,memory}$ in Equation 4.1 to estimate the expected memory queue length(Eq. 4.4a). Then, we predict the memory-bound execution time (in terms of number of processor cycles) as in Eq. 4.4b where f_{core} is the frequency of the processor core.

$$N'_{i,q} = N'_{i,memory}/X'_{i,compute} * T_{max} \tag{4.4a}$$

$$X'_{i,memory} = N'_{i,memory} * T(N'_{i,q}) * f_{core} \tag{4.4b}$$

Last, the predicted execution time for interval i is taken as the worse among the two (memory-bound and compute-bound):

$$X'_i = max(X'_{i,compute}, X'_{i,memory}) \tag{4.5a}$$

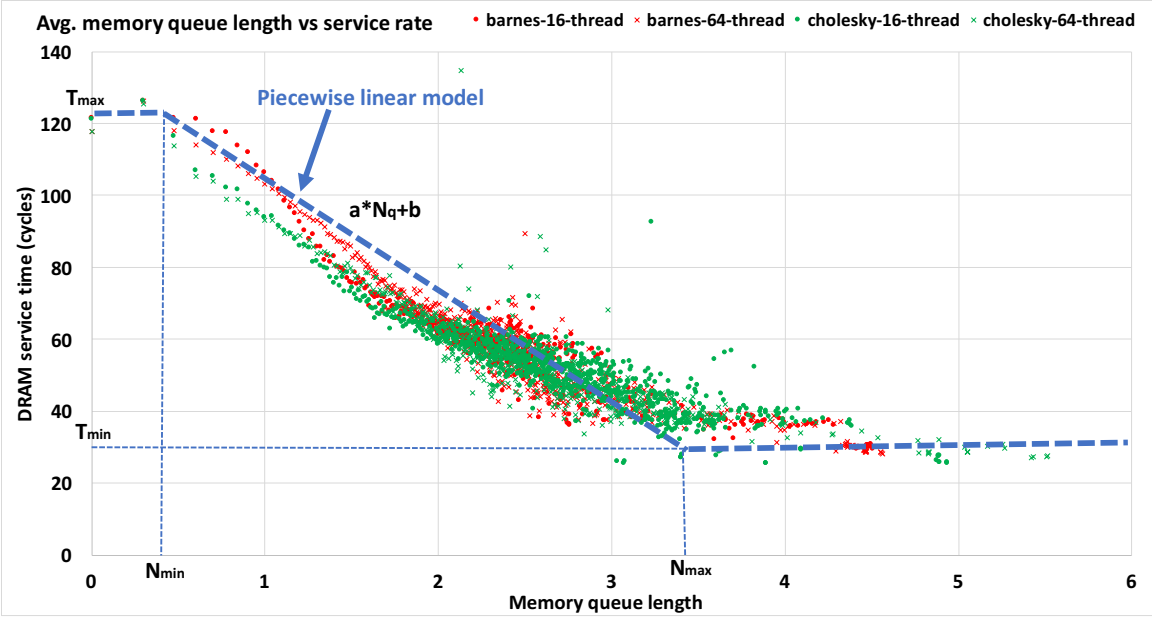

Figure 4.4: Average memory queue length vs average service time

4.1.5 Synchronization Latency

One major performance scaling limiter is barrier synchronization, whose overhead can be broken down into barrier wait (load imbalance) and barrier latency [69]. Our proposed prediction scheme tackles load imbalance by modeling the maximum-instruction-count among threads. For barrier latency, prior work has shown that it increases linearly with the number of threads in log scale, so we use a linear model for it [70]. The overall cycle-count prediction for the application under thread scaling is shown in Eq. 4.6

$$\sum_{i=1}^{M} X_i' + barrier\_delay(N') \tag{4.6a}$$

4.1.6 Cross-Input Prediction

For cross-input prediction, we model how the number of instances for each barrier phase changes with input size, as well as how model parameters in each instance of a barrier phase change with input size. Figure 4.5 shows how the number of barrier phase instances and the

per-barrier-phase-instance instruction count change in *Lu* as its input changes. An instance of a barrier phase starts at some barrier X (or thread-create) and ends at some barrier Y (or thread-join), and we use letters A through E to represent the PC-addresses of the barriers in this application. As shown, when the input size increases, the number of instances increases for some barrier phases, while for others it remains constant. Additionally, the per-instance instruction count can change across instances of the same barrier phase during one run in a significant but predictable way, and the parameters needed for predicting this change can have their own relation to input size. Thus, we model how the number of instances for each barrier phase increases with input, and how instruction count and other parameters of each barrier phase evolve over instances in a single run, and how the input size affects this evolution. We only consider simple models, e.g. constant, linear, etc. because they have few parameters, so they only need a few data points to estimate those parameters. For example, in *Lu* our mechanism identifies the lineal model ($y = ax + b$) to be the best-fitting one for how the per-instance instruction count changes, and estimates the slope (a) and intercept (b) parameters for each input size. It then considers the model for each parameter as input size changes, finding that the slope (parameter a) is best modeled as a constant while the intercept (parameter b) is directly proportional to input size ($b = c * input\_size$). The resulting overall model has only two parameters (a and c) and yet predicts the per-instance instruction count very accurately for various input sizes.

4.2 Evaluation

We evaluate our performance prediction technique using SESC [63], a cycle-accurate architectural simulator. To accurately model the memory system, we replaced the simple memory model in SESC with DRAMSim2 [71], a cycle-accurate detailed memory simulator. We evaluate core setting from 1-core up to 256-cores, with a 16KB instruction/data L1 cache per core. The L2 cache is a distributed shared last-level cache, so each core has a slice of the L2 cache and a router for the packet-switched 2D mesh network-on-chip (NoC).

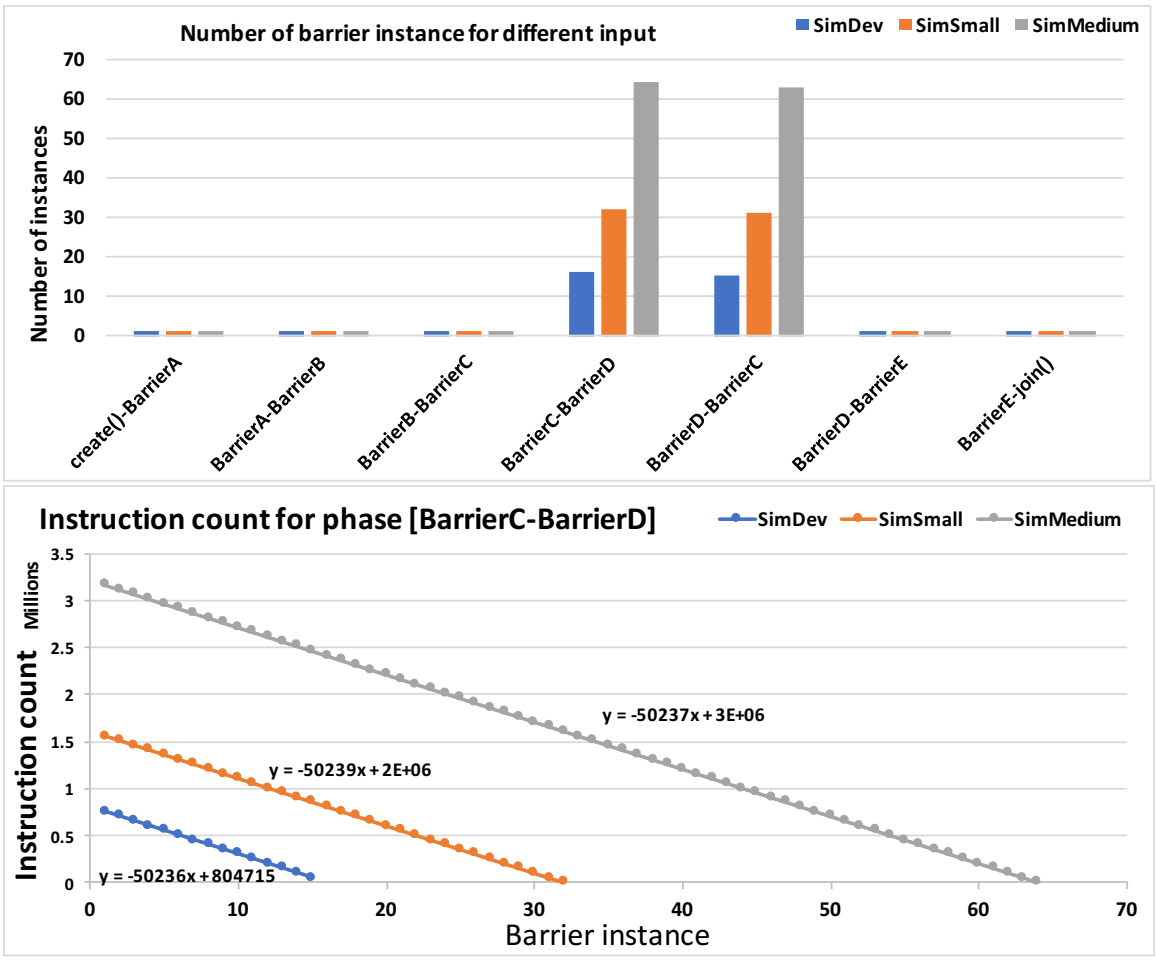

Figure 4.5: How barrier phase scale with input for *Lu*

We model the NoC using Booksim [64], a cycle-accurate NoC simulator that we integrated into SESC. Table 4.1 list the overall configuration for our evaluation.

4.2.1 Speedup prediction

We selected 17 applications from both the Splash [28] and PARSEC [65] benchmark suites. All benchmarks are complied with the GCC 4.6.3 compiler suite using -O3 optimization. All applications are evaluated using 3 inputs: *SimDev*, *SimSmall*, and *SimMedium* input, and only changing the number of threads parameter for each run. We omitted *FMM* and *Facesim* due to very long execution times (note that our evaluation requires a number of simulations for each benchmark), and *Freqmine* was omitted because the way it uses the

Table 4.1: Summary of system configuration

Cores frequency	2.66GHz
Cores	1/2/4/8/16/32/64/128/256
Issue width	2
L1 inst/data-cache	16KB/16KB
L2 cache size	8MB (total)
NoC Network	Mesh
NoC Router	3-stage
NoC Link	128 bits
Number memory channel	4
DRAM	DDR3-1333

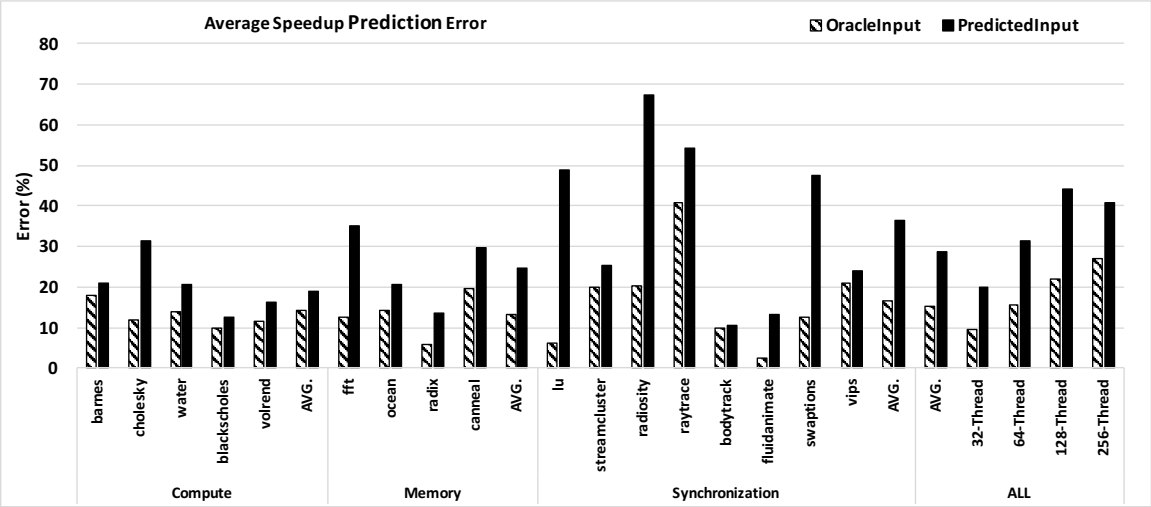

Figure 4.6: Average speedup prediction error

OpenMP programming model was causing our simulations to not complete due to limitations in the simulator's system-call emulation infrastructure.

For all performance prediction models, we collected data for 1-, 2-, 4-, 8-, and 16-threaded runs and used that to train the prediction model, and then we use the resulting trained model to predict parallel speedups for 32-, 64-, 100-, 128-, 196- and 256-thread runs and compare the predicted speedups to speedups obtained from cycle-accurate simulation. We divide each barrier phase into intervals such that each interval contains at most 1% of the total instruction. These results are shown as *Predicted Inputs*. A prediction has two main sources of error: the error introduced by using model parameter values from lower-thread-count runs in high-core-count predictions, and the error introduced by the model itself. This includes the fact that the model may not capture the exact relationship between

the model parameters and execution time, and the fact that the model may not even include some parameters and effects that do affect actual performance. To distinguish between these two sources of error, we also show the accuracy results for *Oracle Inputs*, where the model is allowed to use instruction counts and memory requests from the same simulation run whose performance it is trying to predict. This removes the error that comes from parameter value prediction and leaves only the error introduced by the model itself.

Figure 4.6 shows the average prediction error for all the studied benchmarks. In addtion, we also show the average error for 32 to 256 threads and the overall average error. For benchmarks that require the number of threads to be a power of 2 (*Ocean*, *FFT* and *Radix*), we omitted 100-threaded and 196-threaded speedups from the average for these benchmarks. On average, our proposed model results in an average error of 27%, where 15% error is caused by the simplicity of the model itself and the rest of the error is attributable to imperfect prediction of model parameters. We consider this to be a good result, given that the model is using runs with up to 16 threads to predict performance of runs up to 256 threads, i.e. the performance prediction is for a 16-fold increase in thread count.

For memory intensive application such as *FFT*, *Ocean*, *Radix*, and *Canneal*, our model was able to achieve an average error of less than 24%, with only 13% error with *Oracle Inputs*. This indicates that our simple DRAM model is still effective in predicting the memory system's congestion.

For compute-intensive applications, such as, *Barnes*, *Cholesky*, *Water*, *Blackscholes*, and *Volrend*, our model produces an average error of 18%. Without accounting for the scaling of per-thread instruction count this error would be much larger. Additionally, by modeling the per-thread-maximum rather than average per-thread instruction count, we were able to take into account the uneven distribution of instructions among threads - the change in the max-to-average ratio of instructions per thread ratio varies from 1.1X to 32X in these applications, which underlines the importance of accounting for the work

imbalance among threads. Our model does so by modeling the max-instruction-per-thread.

Finally, several applications that are neither memory- nor compute-intensive: *Lu*, *Streamcluster*, *Radiosity*, *Raytrace*, *Bodytrack*, *Fluidanimate*, and *Vips*. These applications are either barrier-intensive or lock-intensive applications. For barrier-intensive applications such as *Lu*, and *Streamcluster*, our model with *Oracle Inputs* takes into account the increase delay of barrier operations, which benefits the prediction accuracy. However, *Lu* suffered an accuracy loss due to load imbalance, which was not shown in small thread counts. Other applications that are lock-intensive does not perform well since our model does not directly model the overhead of lock contention, it performs sub-optimally for these applications. Note that *Fluidanimate* performs relatively well compared to other lock-intense applications. This is although because although *Fluidanimate* are lock intensive, it has very little lock contention. On the other hand, the barrier wait time increases due to imbalance workload distribution among each thread, which our per-thread-maximum instruction trend was able to capture.

4.2.2 Estimation of Optimal Thread-Count for an Application Performance

One of the uses for an application's parallel speedup predictor is to estimate the optimal thread count for maximum performance. To produce such an estimate, we utilize the extended Amdahl's law [72] model, which assumes the program can be divided into a serial section P_{serial}, a parallel section $P_{parallel} = 1 - P_{serial}$, and overhead $P_{overhead}$. For simplicity, we assume the overhead is linear with the number of threads, thus the overall parallel speedup for N threads can be modeled as

$$Speedup(N) = \frac{1 + P_{overhead}}{P_{serial} + (1 - P_{serial})/N + P_{overhead} * N}$$

and the optimal thread can be predicted using

$$N_{optimal} = \sqrt{\frac{1}{P_{overhead}} - \frac{P_{serial}}{P_{overhead}}}$$

. We obtain the parameters for this extended Andahl's law model using nonlinear least square regression fitting of parallel speedups, and then we obtain the optimal thread count by solving for the maximum on the extended Amdahl's law curve.

As an "ideal" reference, we first obtain the extended Amdahl's Law curves and the corresponding optimal thread counts for *Actual*, which uses speedup points obtained from 1-thread to 256-thread simulations. Another reference we use is *Actual[1:16]*, which uses speedups from 1-thread to 16-thread simulations to perform the extended Amdahl's law curve fitting – this reference corresponds to using the extended Amdahl's law itself as the model that is trained at low thread counts and then used to predicting speedups at larger thread counts. Finally, *Predicted Input* which uses actual speedups from 1-thread to 16-thread simulations, trains our model using data obtained from those same (1-to-16-thread) simulations, and then uses our model to predict the speedups for the remaining thread counts (32-to-256-thread).

Figure 4.7 shows the actual and our model's predicted speedups, as well as the three fitted extended Amdahl's law curves, for the *Radix* benchmark from Splash-2. As shown, naively fitting the extended Amdahl's law using low-thread-count points data results in significantly over-estimating the potential speedup, mainly because the $P_{overhead}$ value produced by regression at these low-thread-count points is grossly underestimated. In contrast, from these same low-thread-count runs our parallel performance prediction scheme is able to produce relatively accurate speedup estimates for high-thread-count configurations, allowing extended Amdahl's law curve regression to much more accurately predict the thread count at which the parallel speedup peaks.

Figure 4.8 shows the error for $N_{optimal}$ estimate. Naively predicting the optimal thread

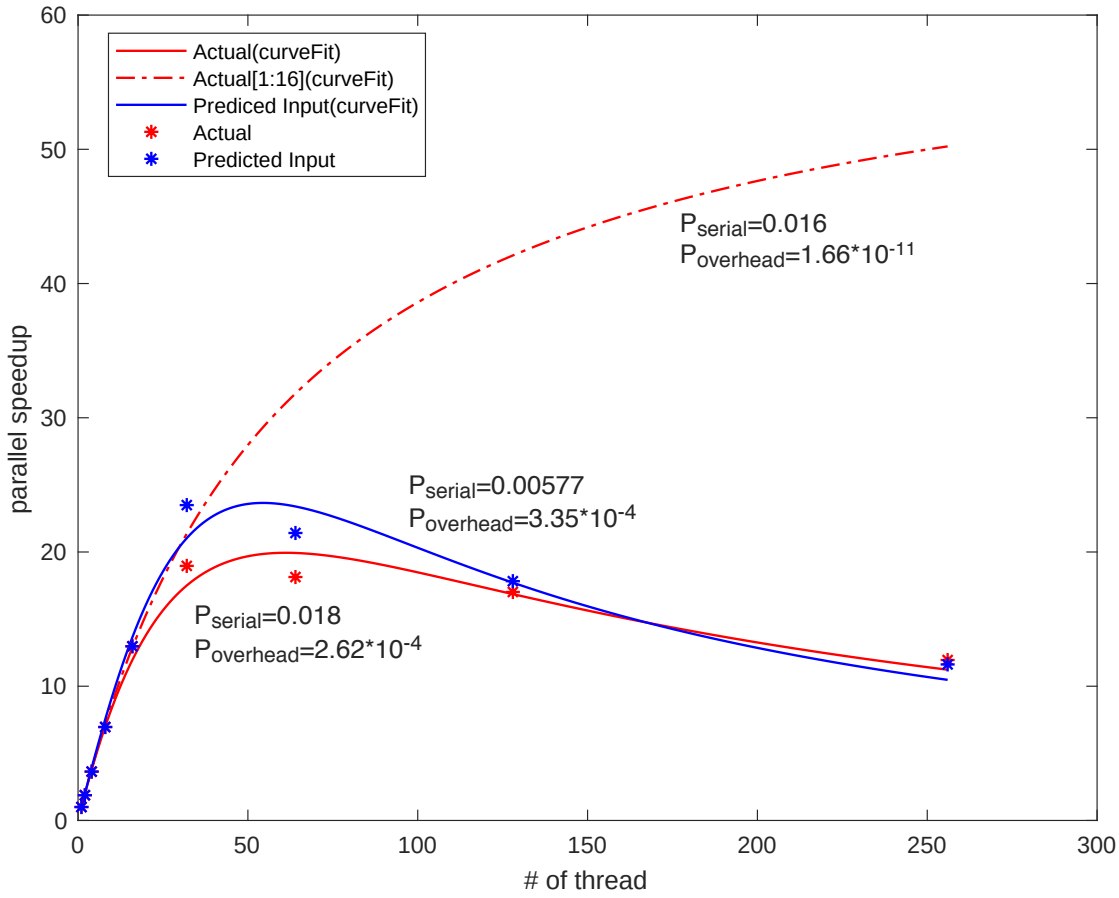

Figure 4.7: Curve fitting for Radix using extended Amdahl's law

with *Actual[1:16]* result in an average error of 90%. For compute and memory intensive application, the average error of $N_{optimal}$ using our scheme is only 23%. However, the prediction error is larger, 46% on average, when we include applications that are synchronization-bound. Since our prediction model does not model synchronization overheads, it over-estimates the speedup on these applications and thus under-estimates $P_{overhead}$.

4.2.3 Error breakdown

Figure 4.9 shows the breakdown of the speedup prediction error of our approach. The error is broken down into four components: *Model*, which is the error that results even when using oracle parameters in our model, *ErrorAccu.* which is the accumulation of the error due to using parameter estimates obtained by thread-counts that are more than

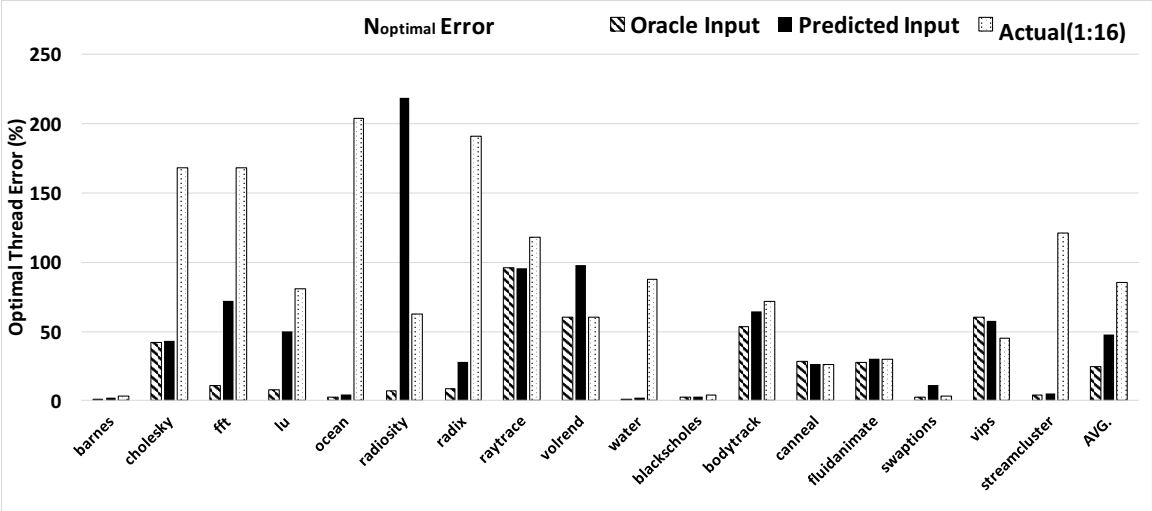

Figure 4.8: Error for $N_{optimal}$

single thread-count-doubling removed from the one we are trying to predict, *InstPredict* is the error introduced by imperfect prediction of the per-thread instruction count, and *MReqPredict* is the error introduced by imperfect prediction of the number of memory requests (LLC misses).

While no single component of the error is strongly dominant, the largest component of the error is the *Model*, which contributes a 9.8% error on its own. This error is a consequence of the model's simplicity of the model, e.g. its lack of explicit modeling of memory level parallelism(MLP), synchronization overheads, etc. The next largest contributor is *ErrorAccu*, which contributes an additional 8.3% error. This error is a consequence of accumulation of error when the core counts of training and prediction differ a lot – recall that we train using 1-to-16-thread runs and then predict performance of 32-256-thread runs, i.e. there is a 16-fold difference in the number of threads between the training runs and predicted runs. Note that *radiosity* shows the largest error caused by this accumulation. Both the model and the accumulation error in this application are mainly caused by not modeling lock contention - lock contention increasingly degrade performance as the thread count is increased in this application. Training at a thread count that includes some synchronization overhead allows the model to capture some of the effects of synchronization overhead.

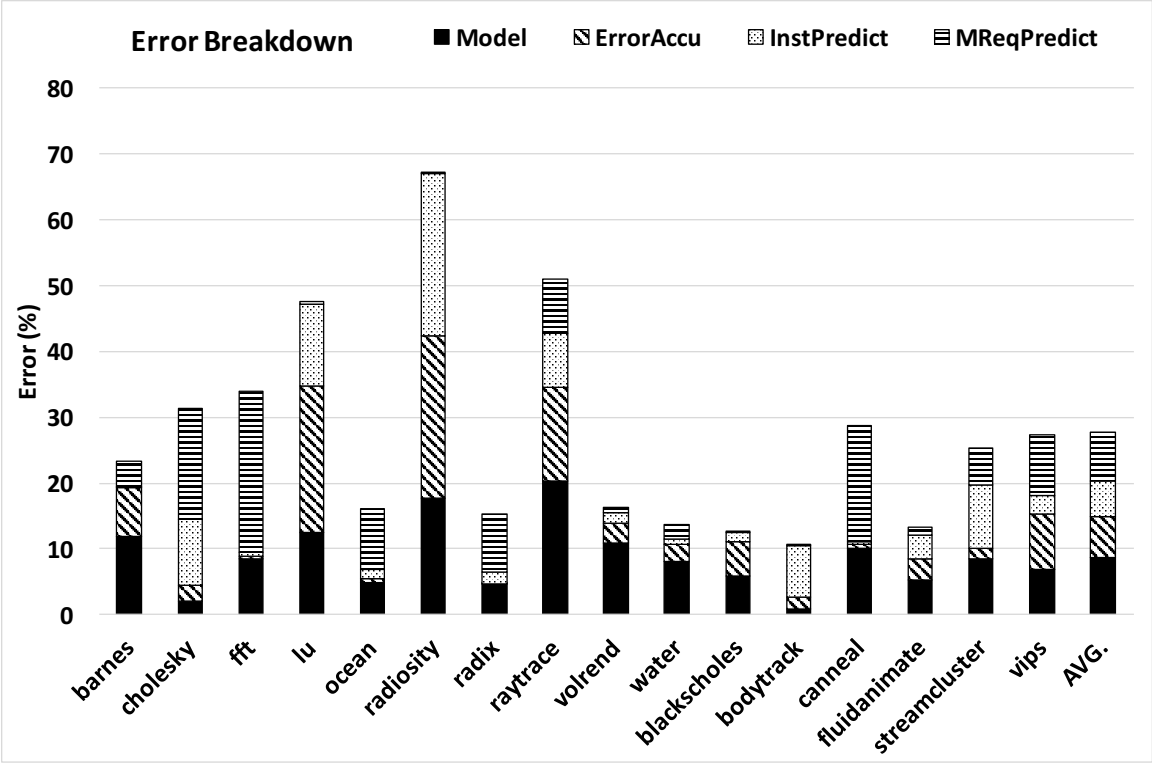

Figure 4.9: Breakdown of speedup prediction error

However, as the prediction is removed further (in thread-count) from training, these effects become more prominent. Since our model absents an explicit model of synchronization overheads, it fails to capture that which results in a large accumulation error.

The next error component in terms of magnitude is *InstPredict*, which contributes an additional 8.3% error on average. The largest instruction-count prediction error is in *radiosity*, where the application utilizes work queue and allows work stealing from different threads, which prevents the instruction count from following its normal trend. Finally, *MReqPredict* contributes an additional 6.3% of error. The largest contributor to this error component is *FFT*, where this error is caused by the CRD stops shitting at 128-threads. The stopping of CRD shifting after certain thread count was also noticed by prior work[29].

4.2.4 Interval Analysis

To understand how the number of intervals affect our performance prediction model, we vary the number of intervals used in each application from 100% (entire barrier phase is one interval) to 10% and then to 1% per interval per application run. Figure 4.10 shows how the average prediction error changes as the number of intervals per application changes. For clarity, we only show benchmarks that have at least 3% improvement when increasing the number of intervals, but the average shown is across all benchmarks.

On average, the prediction error is reduced by 5% when using 1%-intervals compared to using a single interval for each barrier phase. For memory intensive applications, such as *FFT* and *Radix*, the prediction accuracy increases with the number of intervals. Intuitively, with more intervals each interval is shorter and thus more likely to capture a homogenous behavior in terms of the memory request rate. Conversely, longer intervals are more likely to include both bursts of memory accesses and "quiet" periods, where the memory access time estimates that are based on the interval's average memory access rate fail to capture the dramatic increase in memory contention during bursts. For compute-intensive applications, however, prediction accuracy is largely unaffected by the number of intervals. Interestingly, for *Cholesky*, the error actually increases when increasing the number of intervals. This is due to the over-partitioning of the application phase behavior.

4.2.5 Input Scaling

Figure 4.11 shows the average prediction error changes when only using small threads (1-thread to 16-thread) and small input (SimDev, SimSmall) results, to predict large threads (32-threads to 256-threads) with large input (SimMedium). On average, the prediction error only increases by 2% on average when compared to only performing thread scaling prediction. This shows the effectiveness of leveraging the structural change in barrier phases to predict across different input size. For *Water*, the performance degraded significantly under input scaling. This is because the instruction scaling trend were not able to be sampled

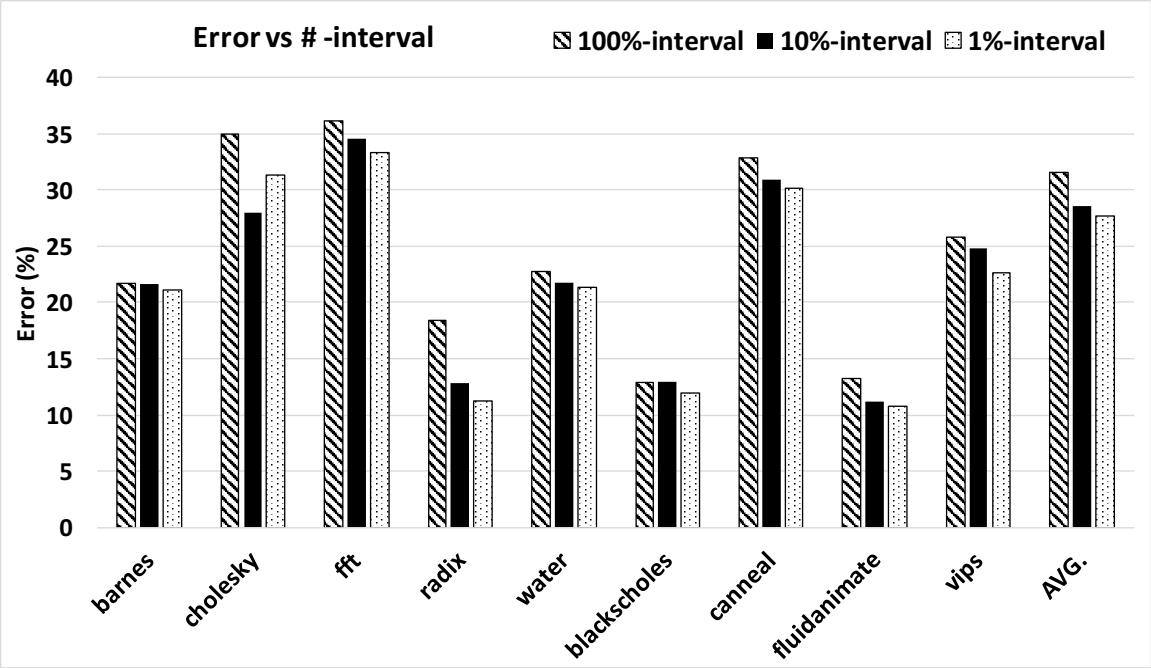

Figure 4.10: Number of intervals on the effect of average prediction error

correctly with just two data points (SimDev and SimSall), since with two data points we can only model the instruction scaling trend with a linear model. We believe with more sample points; the prediction result will improve with better prediction on the instruction scaling trend.

4.2.6 Core Frequency Scaling

In previous evaluation, we have assumed the same core frequency when scaling the number of cores. However, core frequency is often reduced when increasing the number of cores due to a limited power budget. Hence, we want to study how our scaling prediction scheme can be applied in the context of core frequency scaling. We evaluate the prediction scheme when training with 1 to 16 cores, with each core running at 2.66GHz, and then predicting performance for 32 to 256 cores with each core running at only 1.33GHz.

In order to account for frequency scaling, the only parameter we need to modify is f_{core} in Equation 4.4b. Note that $T(N_q) * f_{core}$ represents the expected DRAM service time in terms of the number of core cycles, therefore, when clocking the cores at half the

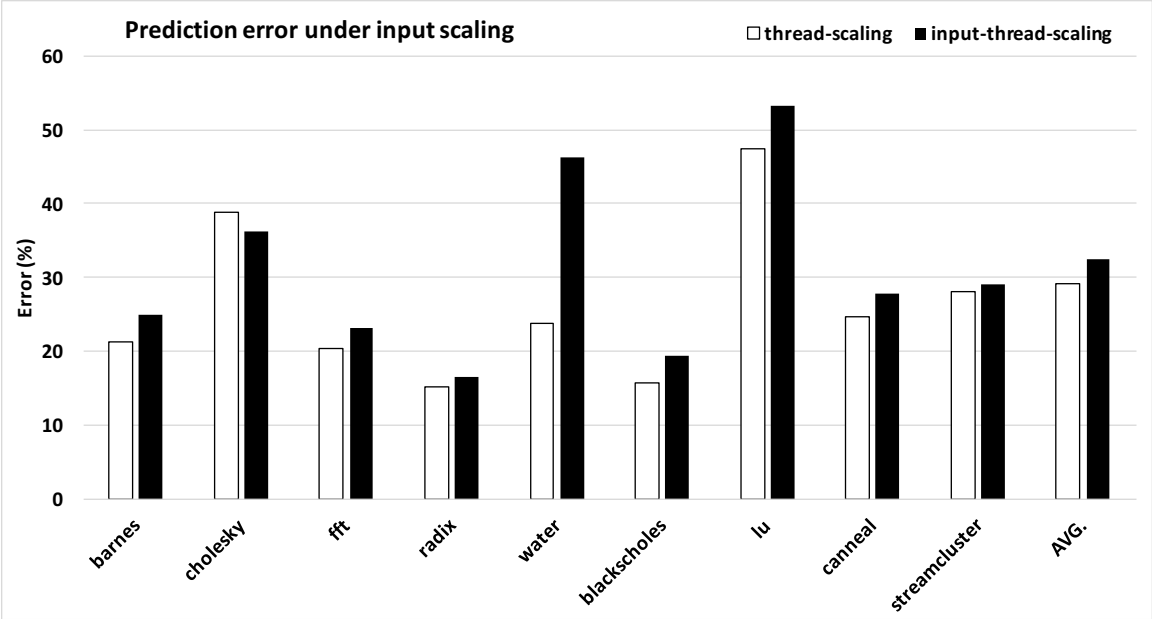

Figure 4.11: Average error for thread scaling and input-thread-scaling

frequency, the DRAM latency in core cycles is cut in half. Figure 4.12 shows the average prediction error in this scenario. For simplicity, we only show individual-application results for memory-intensive and compute-intensive applications, but the average is still calculated over all benchmarks. The results show that the error is very similar to the error observed without frequency scaling. This implies that our model is sufficient to account for the effects of core frequency scaling, without introducing a significant additional error.

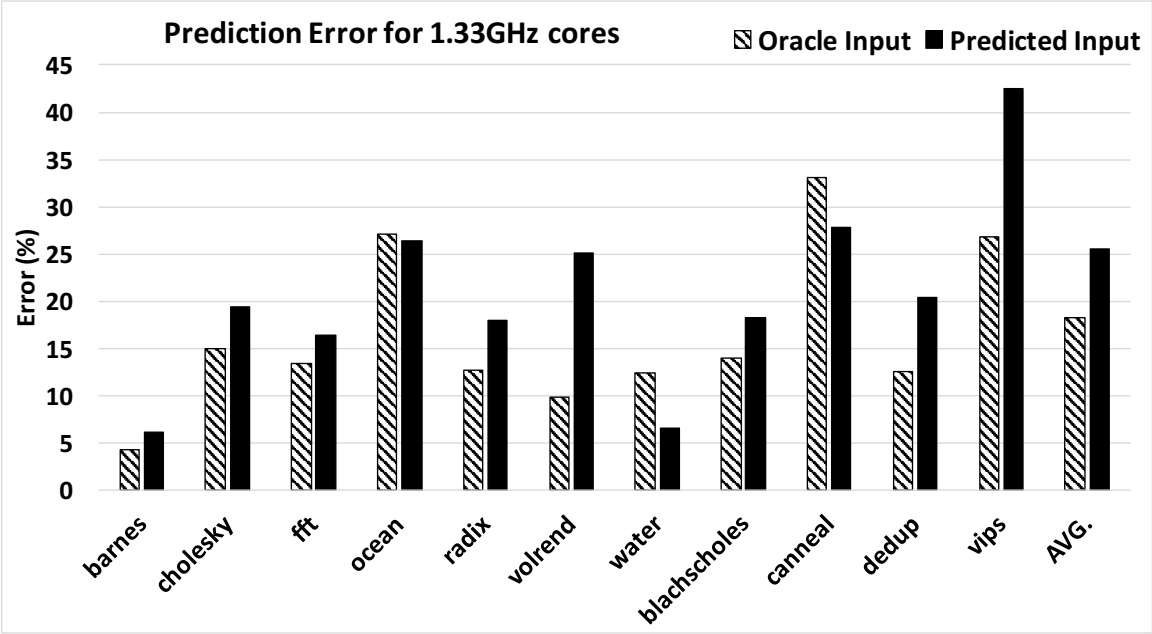

Figure 4.12: Prediction error under core frequency scaling

CHAPTER 5
LOCK CONTENTION PREDICTION USING PC-BASED STATISTICAL MODELING

In this chapter, I present a lock contention model that records statistics for each static location in the code (PC address) at which the lock is acquired and predict how the lock contention for each static lock function call will increase when scaling the number of threads. The model captures program characteristics of multi-threaded applications from few-threaded runs, builds a statistical model and predicts how the model would change with thread count.

5.1 Model Structure

5.1.1 PC-based prediction model

To account for different parts of the application having varying lock access pattern and critical section behavior, we propose to model the lock contention for each lock function call (lockPC) separately. In addition, we partition the program execution into parallel sections and model the lock contention scaling of each lockPC, separately. In this paper, we consider a parallel section to be the part of the program that executes between one global synchronization (e.g. a barrier, fork create/join) and the next one. Note that each static parallel section (code that follows a static PC where barrier() is called) can have multiple dynamic instances. For simplicity, we will refer to each dynamic instance of a static parallel section as "parallel section", and explicitly state a static parallel section otherwise.

Figure 5.1 illustrates how we perform lock contention prediction. For each parallel section, we first identify all lockPCs (Lock#). Then, we collect statistics (inter-arrival rate, critical section size, etc.) and use well-known statistical models to approximate each lockPC

characteristics. To predict the lock contention when scaling the number of threads, we first collect statistical behaviors using small thread counts, and then build a regression model to predict how each variable change with increasing number of threads. We then predict the lock contention for each of the lockPC, separately, and aggregate the total lock contention to calculate the average lock wait time for a single thread. Note that each lock/unlock pair represents a critical section, and that we do not distinguish between nested locks or not.

One might argue that lock contention happens due to multiple threads acquiring the same lock variable, therefore why not mode the lock contention for each lock variable. The reason is twofold. First, the amount of lock variables changes with the number of threads and input size. Therefore, it is much difficult to model the scaling of lock contention when the number of locks variable varies. In order to do so, it would require grouping the locks into similar behavior, and then predict the scaling trend of each group of locks. Luckily, using the lockPC to group lock behavior together is a very feasible mechanism to identify similar behavior on different lock variables. For lockPC that uses a thread-indexed lock array, it will be easier to predict how the lock contention will scale. Second, in order to take into account threads accessing the same lock variable from different lockPCs, we propose a model refinement (Section 5.2) which merges lockPCs that accesses similar sets of lock variables and model the lock contention together.

5.1.2 Arrival rate

Arrival rate is the first-order representation of how often threads arrive to a certain lockPC, it also is one of the important contributing factor in how severe the lock contention is. When threads enter a parallel section, either from *thread_create* or from the release of a barrier, they often begin in sync and tends to execute in similar code regions. Once threads have been executing longer in the parallel section, their execution will be less in-sync and thus the arrival rate will then be dictated by the control paths of the program. Either way, the end result is thread arrival rate often correlates with the lockPC.

Figure 5.2 shows the average inter-arrival time for different lockPCs, with the y-axis in log scale. As shown, different lockPCs experiences dramatically different inter-arrival rate, with a difference up to 1000x. In this example, PC1 is the first lockPC each thread will encounter after leaving the barrier, therefore threads end to arrival in bursts. For PC3/6, it is within a for loop with large loop body, therefore have a larger inter-arrival time.

To collect the arrival rate, we profile the application by recording the cycle time and PC address before a thread enters the lock function call. The profiling can be done locally to each thread in order to reduce the profiling overhead. Then, the per-thread lock arrival timestamp from all active threads are aggregated in order to create the overall sequence of thread arrival to a particular lockPC. With that, we can calculate the inter-arrival time between each thread arrival and create the histogram of inter-arrival time, which is used as the inter-arrival time probability model for determining the lock arrival rate.

To predict the changes to the inter-arrival rate when scaling the number of threads, we first identify a suitable statistical model to represent the inter-arrival time. Prior works for database systems have often assumed a Poisson arrival rate, which results in an exponential distribution model for inter-arrival time. Our evaluation shows that for some lockPC, exponential distribution model does match the inter-arrival time histogram, with others needing more complex models such as the inverse-chi-squared model. For simplicity, we will assume the exponential probability model. Since the exponential probability model can be represented with just one parameter (rate parameter λ), we can now use regression models to model the change of the rate parameter λ with varying thread count. Figure 5.3 shows the actual and predicted rate parameter for a particular lockPC. As shown, the predicted model (a Power trendline) can predict how the rate parameter will change with increasing thread count. Note that the main advantage of representing the inter-arrival histogram with a well-known probability model is to reduce the model parameter space. Therefore, we can reduce the large histogram to just 1 or 2 parameters that can be easily predicted using limited data points.

5.1.3 Critical section

Critical section size represents how long a thread will hold a particular lock and is often one of the main contributing factor in how severe the lock contention is. The longer the critical section is, the more likely a thread will arrive to the lockPC to access a contended lock. Therefore, it is critical to model the critical section length in order to properly model the lock contention.

While prior works have also taken into account the importance of modeling the critical section, their model often assume a constant latency for each critical section. In addition, prior model often assumes all critical section are equal, meaning they tend to model how likely critical sections conflict simply by looking at the total amount of time a thread spends in a critical section. However, simply assuming a constant critical section latency for all critical section is not accurate. Figure 5.4 shows the critical section length for various lockPC. As shown, different lockPC will exhibit different degree of critical section size, with some in 10s of cycle and others in 10000 cycles. The large magnitude different in critical section length emphasis the importance of modeling each lockPC separately, in order to accurately capture the effect of critical section size on the lock contention.

Second, the minimum and maximum of critical section size can vary up to an order of magnitude, therefore, using a constant model underestimates the severity of lock contention. In order to better model the critical section size, we propose to utilize a prato distribution to model the critical section for each lockPC. One important aspect of prato distribution is the long tail of probability. This allows us to model the sudden increase in critical section length with an exponentially decreasing probability. Note that although most critical section have a constant number of instructions (eq. increment a shared counter), some do have varying instructions for each instance. One example is when updating a tree structure, the amount of work needed to before is data-dependent. For such PCs, we proposed to utilize a gaussian distribution to model the critical section size. This is the case for lockPC5 in Figure 5.4. As shown, the average latency of the critical section sits relatively

in the middle of the distribution, instances are equally likely to increase or decrease in latency. In summary, our critical section latency model can be categorized into two cases, as shown in Eq. 5.1

$$CS_model = \begin{cases} CS_{Tavg} * Prato(\alpha), & CS_{inst} \text{ constant.} \\ gaussin(CS_{Tavg}, CS_{Tstd}), & \text{otherwise.} \end{cases} \tag{5.1a}$$

To profile the critical section for each lockPC, we propose to collect both the cycle count and instruction count when we first enter a critical section (after acquiring the lock), and also before we exit the critical section (before unlocking). Note that this too, can be done separately for each active thread. Hence, profiling can be done locally to each thread in order to reduce the profiling overhead. One could also reduce the overhead for profiling by sampling the behavior of each lockPC, simply by only recording for a certain number of instances.

To predict the changes to the critical section size, we propose to use regression modeling technique to model the change of the parameter for either the PratoDistribution model, or the gaussian distribution mode. Both models have 2 paraments, one representing the average, and the other represents the spread, either through the α parameter for prato distrubtion or the σ for the gaussian distribution model.

5.1.4 Lock access histogram

One of the main benefit of modeling lock contention for each lockPC is the easiness of predicting the lock access histogram. Lock access histogram represents the number of accesses on each lock variable. For lockPC that access a single lock variable, the lock access histogram is simply 100% on 1 lock variable. However, for programs that uses fine-grain locking technique to reduce lock contention, the number of lock access is spread across multiple lock variables, with varying distribution.

Figure 5.5 shows the lock access histogram for various lockPCs. As shown, some lockPC have a single lock variable, thus shows a spike of 100% at the beginning. Other lockPC have varying degree of spreading of lock accesses, such as a hot-spot histogram with 1 lock accounting for the majority of lock accesses, or a uniform histogram where lock accesses are spread evenly across various lock variables.

One important aspect of this is that the lock access histogram is inherently a result of the program characteristic. Therefore, lock access histograms tend to change in a structural manner. For example, for program that uses lock array and is indexed with thread id, the histogram tends to be uniform access histogram and the number of lock variables scales with the thread count. On the other hand, lock histogram with 1 hot lock variable will also tend to exhibit the same behavior while scaling the number of threads.

In addition to histogram, the total number of lock access also scales with thread count in a structural manner. For lockPC that have the number of lock access associated with input size, the total lock access remains constant when scaling the number of threads. On the other hand, for lockPC that have a constant per-thread access rate, the total lock access will then scale according with thread count.

We propose a two-way lock access histogram prediction scheme which leverages the structural change of the total lock access count and lock access histogram. First, predict how the total number of lock access and the total number of lock variables scales with thread count. Note that due to the structural behavior of lock access count, simple linear or power model is sufficient to model the scaling of total lock access and lock variable. To model the histogram change, we directly predict the profile change when scaling the number of lock variables. Figure 5.6 shows how the histogram prediction scheme works. By modeling the shifting of histogram through the scaling of lock variables and the envelope change, we are able to track how the lock histogram profile shifts with thread count.

5.1.5 Lock handoff model

Liang [32] have shown that lock handoff latency increasing with thread count, whereas Boyd-Wickizer [46] have discussed the importance of modeling lock handoff latency when attempting to model lock contention. There are two take away for lock handoff latency. First, lock handoff latency strongly depends on the lock algorithm. The latency profile for MCS locks and spin-locks exhibits dramatically different characteristics when the lock contention increases with thread count. Second, the increase in cache-to-cache transfer latency also exasperates the lock handoff latency.

While prior works have discussed how to model the lock handoff latency, we proposed to utilize simulation technique to model and predict the average lock handoff latency. By utilizing a simulation model, our model can easily adopt the lock handoff latency to different types of system and lock algorithm. Note that unlike running simulation for a full benchmark, running simulation on a small kernel which evaluates the lock handoff latency incurs relatively low overhead.

5.1.6 Overall model

In summary, our PC-based lock contention model predicts the overall lock contention by modeling the lock contention for each lockPC separately. For each lockPC, we predict the lock contention with a probability model that consists of 4 paraments - inter-arrival rate, critical section latency profile, lock access histogram and average lock handoff latency. For inter-arrival rate and critical section latency, we first profile and collect the statistical profile for the two parameters and match it to a well-known mathematical model such as exponential distribution. Then, we utilize a regression model to predict how the model parameter changes with increasing thread count. For lock access histogram, we utilize regression models to predict the total number of lock access and total number of lock variables, then predict the profile change with increasing thread count. For average lock handoff latency, we utilize a detail simulator to extract the average handoff latency when

increasing the number of threads.

To calculate the lock contention, we utilize Monte-Carlo simulation technique to simulate the average lock wait time for each lockPC along with the predicted probability model for each lockPC characteristics. Using Monte-Carlo simulation enables us the flexibility to have different probability models for various input parameters, such as lock access histogram. Although closed-form solution can quickly calculate the expected average latency, it requires each parameter to be represented as a well know mathematical model, which in result limit the accuracy of the prediction. Note that since we are using Monte-Carlo simulation to calculate the expected lock contention, our model can easily incorporate various lock scheduling scheme such as first-come-first-server which represent queue-based lock algorithm, or a random scheduling scheme which matches a spin-lock algorithm.

5.2 Model Refinement

5.2.1 Merge lockPC

One of the challenge for using PC-based lock contention model is that lock contention happens not on lockPCs, but on lock variables. Hence, even if threads access the same lock variable from different lockPC, it still results in lock contention. For example, *Raytrace* has 1 for loop with 3 lockPCs, each accessing a single and common lock variable. Therefore, threads arriving to either one of the 3 lockPCs will result in accessing the same lock variable.

To handle lock variables being accessed from different lockPCs, we refine our model to identify and merges lockPCs that have a significant overlap in terms of which lock variable they are using often into a single unified lockPC. For each lockPC, we define *lockSignature* as the set of top used lock variables that accumulatively accounts for 95% of lock accesses. In result, two lockPCs that have the same *lockSignature* are considered mergeable (they mostly access the same set of lock variables). Note that we use a empirical threshold of 95%, which can be tuned to adjust how aggressively to merge lockPCs. Figure 5.7 gives

an illustration of how *MergePC* works. As shown, since Lock2 and Lock3 have the same lockset, we merge the two lockPC together.

Note that for lockPCs that are grouped together, their corresponding statistical model are still considered separate, but instead are evaluated together as a whole. Figure 5.8 shows how we evaluate the lock contention for lockPCs that are grouped together. Each lockPC still maintains its own statistical model for various parameters (#lock access, inter-arrival rate, etc.). However, whenever a thread releases a lock, it will determine the next lockPC to arrive at according to the distribution of total number of lock access ($N_1 + N_2$) for each lockPC. Once the next lockPC is determine (eq. Lock2), all related stats (inter-arrival time, critical section length, etc) are determined according to the histogram of that particular lockPC.

One advantage of keeping the statistics of each lockPC separate is to simplify the prediction of model parameters. Since each lockPC behavior changes with thread count in a structural way, predicting the model parameters separately and merging them in the simulation allows us to continue to use simple regression schemes to predict the model change while still achieving good accuracy.

5.2.2 Merge Parallel Section (PS)

Statistical modeling and prediction relies on sufficient data to properly represent the underlying behavior. For lock contention modeling, this means for each lockPC, there are sufficient amount of lock accesses. For lockPC that are within *For* loops, this is not an issue since the for loop ensures each thread participate in lock access multiple rounds. However, for lockPC that are not within a for loop, the number of lock access are not sufficient since the number of lock access only scales with the number of threads.

To obtain sufficient data points, we propose to combine the statistical behaviors of parallel sections that are from the same static parallel section in order to increase the available data points for statistical modeling. Since we are combining dynamic parallel sections

from the same static parallel section, each dynamic parallel section can be considered as a sample of the intrinsic behavior of the static parallel section. This is particular useful when evaluating with small thread count, since our model relies on capturing the statistical behavior using small thread counts, mergePS is critical to improve the accuracy of lock contention prediction. Note that for some application, the same static parallel section may have multiple code path and depending on the loop iteration (eq. the first or last iteration), threads will take a different code path. For these cases, we do not merge the parallel sections that differs drastically (eq. the number of lock access, instruction count, etc.), but instead leave them separate.

Figure 5.9 shows the inter-arrival time for *Ocean*. As shown, since the lockPC is accessed once by each thread within each phase, there are not sufficient data points to provide a good statistical modeling for the inter-arrival histogram. However, if we merge data points from different parallel sections, we can obtain enough data points to capture the statistical behavior.

5.3 Evaluation

We evaluate our performance prediction technique using SESC [63], a cycle-accurate architectural simulator. To accurately model the memory system, we replaced the simple memory model in SESC with DRAMSim2 [71], a cycle-accurate detailed memory simulator. We evaluate core setting from 1-core up to 256-cores, with a 16KB instruction/data L1 cache per core. The L2 cache is a distributed shared last-level cache, so each core has a slice of the L2 cache and a router for the packet-switched 2D mesh network-on-chip (NoC). We model the NoC using Booksim [64], a cycle-accurate NoC simulator that we integrated into SESC. Table 5.1 list the overall configuration for our evaluation.

We selected 7 applications from both the Splash [28] and PARSEC [65] benchmark

Table 5.1: Summary of system configuration

Cores frequency	2.66GHz
Cores	1/2/4/8/16/32/64/128/256
Issue width	2
L1 inst/data-cache	16KB/16KB
L2 cache size	8MB (total)
NoC Network	Mesh
NoC Router	3-stage
NoC Link	128 bits
Number memory channel	4
DRAM	DDR3-1333

suites that either shows a sufficient amount of lock contention or has many lock access. All benchmarks are complied with the GCC 4.6.3 compiler suite using -O3 optimization. All applications are evaluated using *SimSmall* input, and only changing the number of threads parameter for each run.

For all performance prediction models, we collected data for 2-, 4-, 8-, 16- and 32-threaded runs and used that to train the prediction model. Then we use the trained model to predict the lock contention for 64-, 100-, 128-, 196- and 256-thread runs and compare the predicted lock contention to the lock contention obtained from cycle-accurate simulation.

5.3.1 Benchmark summary

To understand how each benchmark provide a different lock access scenario, Table 5.2 summarize the lock access pattern and lock contention for the benchmarks that we evaluated.

Raytrace shows the highest lock contention among all applications, which on average spends 71% of execution time waiting for lock. The application has 7 lockPCs, with 2 of them accounting for 70% of lock contention, and 2 other accounting for the rest of 30% of lock contention. The main reason for lock contention is simply due to the average inter-arrival time is roughly similar to the average critical section time plus the lock handoff latency. Therefore, the service rate and arrival rate are equal in such cases. Interesting, the 2 most contented lockPC actually access the same lock variable, therefore our *MergePC*

refinement was able to capture and model the reduction in inter-arrival rate.

Volrend shows the second highest lock contention among all applications, which on average spends 48% of execution time waiting for lock. The application has 4 lockPCs, with 1 of them accounting for 86% of lock contention. Interesting, the lockPC is not the most accessed lockPC, but created a large lock contention due to the small average inter-arrival rate, which is the result of threads arriving to the lockPC synchronously. In addition, the lockPC also shows a large critical section size, which also exasperates the lock contention.

Radiosity experiences on average 34% of lock contention. With 101 lockPCs in the application, most of the lock contention are generated from 2 lockPCs, which accounts for 96% of lock contention. These two lockPCs represents the addition and deletion of work from a shared work queue.

Barnes have 4 static parallel sections, with two of them containing lock accesses. Among the two parallel sections that have lock accesses, 1 parallel section dominates the majority of lock contention and accounts for 7% of execution time. The reason for lock contention is due to the short inter-arrival time, which is a result of having a lockPC within a tight for loop. The two parallel sections have 2 lockPCs and 1 lockPCs, respectively. Unlike other applications where the majority of lock contention is generated from a lockPC with a single lock variable, the hot lockPC actually uses an array of lock variable. However, the lock access histogram exhibits a hot-lock pattern, therefore majority of the lock access are to 1 single lock variable.

Cholesky have 5 static parallel section, with two of them containing lock accesses. Among the two parallel sections that have lock accesses, 1 of them dominate the majority of lock contention and accounts for 18% of execution time. The reason for lock contention is due to large critical section size, with each thread accessing only 1 time. Interestingly, with other lockPCs that have more lock access, the lock contention is very low due to the usage of lock array and the relatively smaller critical section length when compared to the

inter-arrival rate.

Ocean have 5 lockPCs within the application, however, no one lockPC dominates the lock contention. All of these lockPCs, however, exhibits lock contention due the synchronous arrival of threads. Therefore, the difference in lock contention is a direct result of the varying inter-arrival rate for different lockPCs. Note that all of these lockPCs only have 1 access per thread in each parallel section. Hence, *MergePS* was able to aggregate statistics from different dynamic parallel sections and build a more reliable model.

Canneal have 4 static parallel sections, with 1 parallel section containing lock. The parallel section has 1 lockPC and 1 lockAddr, and each thread accesses this lockPC once. Lock contention can achieve as high as 18% of execution time, due to the large critical section that creates high lock contention.

5.3.2 Model prediction result

Figure 5.10 shows the lock contention for all the benchmarks that we evaluated, as well as the predicted lock contention for our PC-based prediction model. The percentage represents the amount of average lock contention that is experienced by each thread for all the parallel sections. *Actual* represents the experienced lock contention wait time for the evaluated runs. *Model(OracleParameters)* uses the histogram collected from each run and represent it has a well-known statistical model as input arguments into our lock contention model. This reflects how accurate our model is at predicting lock contention.

In result, our model was able to predict the lock contention within 7% of actual lock contention. For *Barnes*, our model was not able to capture the relatively small lock contention. This is because we use a lock access histogram to predict which lock variable each thread will access; however, this does not fully represent the behavior of lock variable access experienced in the actual run (there are correlations between accesses of lock variables). *Raytrace* shows the largest error, with 22% of difference to actual lock contention. However, since the lock contention is fairly high for *Raytrace* (up to 70%), our

PC-based model predicted the lock contention as 50%, which clearly captures the large lock contention.

Model(PredictedParameters) uses data points collected from small thread count runs, and uses regression model to predict how each input argument will scale when increasing the thread count. This represents the most common use-case our proposed PC-based lock contention model, which allows us to predict how the lock contention will change with thread count. As shown, our prediction scheme was able to achieve prediction accuracy within 13% on average. *Radiosity* and *Raytrace* contributes to most of the error, which both over-predicts the lock contention due to the rebound of interArrival time. For small thread count, the inter-arrival time follows a power trendline and continues to decrease, however, for large thread count, the inter-arrival time stops scaling and also slightly increases. This non-linear behavior is hard to predict when the training data for the regression model does not exhibit such behavior.

5.3.3 Error breakdown

To understand which component contributes to the increase of error in our prediction mode, Figure 3.9 shows the breakdown of error contribution of each factor for our model. *Simplified Model* represents the error created due to the behaviors not modeled by our PC-based histogram prediction model. This includes errors such as the correlations of lock variable accessed between each lock access. *Approximated Histogram* represents the error due to transforming the actual histogram into a paramterized mathematical model. *Regression Fitting* represents the error due to using simple regression model to capture the correlation of thread count and model parameter. Last, *Insufficient Training Data* represents the error due to using only small threads count as training data for our regression model.

On average, our base model contributes about 3.2% of error in our lock contention prediction scheme, which is also the largest component. These errors are the result of using statistical models to determine the inter-arrival time, lock access, etc. A more sophisticated

model can be used, such as incorporating correlation between previous accesses in order to build a more accurate statistical model. However, this will significantly increase the model complexity, which we believe we strike a good balance between complexity and accuracy.*Insufficient Training Data* contributes the second largest error, which is simply the result of using small thread-counts to train the regression model. In our evaluation, we are using data points up to 32-threads to predict performance for 256-threads. Note that this is an 8X increase in thread count, which results in some characteristic not shown in small thread-count runs.

5.3.4 Refinement analysis

Figure 5.12 shows the lock contention prediction accuracy with and without *MergePC* and *MergePS* refinement. For simplicity, we only show applications that have at least 15% difference in prediction accuracy. For *Ocean*, using *MergePS* was able to increase the prediction accuracy by 8%. This is because for *Ocean*, each thread only access the lock variable once in each barrier phase. Therefore, it is hard to acquire an accurate statistical model with just little of data. However, with MergePhase, we were able to aggregate data points from different barrier phases to construct a better statistical model, hence improve the prediction accuracy. Note that MergePhase not only increases the accuracy for some applications, it also reduces the simulation time since each barrier phase only needs to be simulated once, and the lock contention can simply be multiplied by the number of dynamic phases.

For *Raytrace*, *MergePC* was able to greatly improve model accuracy. This is due to the fact that for *Raytrace*, there is a nested loop with 3 different lockPC, each accessing the same lock variable. Hence, these 3 lockPC needs to be aggregated and modeled together, since threads arriving at either one the of the lockPC will incur the same contention on the same lock variable.

5.3.5 Extended Amdahl's Law

Figure 5.13 shows the lock contention prediction accuracy that extends Amdahl's law to incorporate lock contention modeling [45]. As shown, the simplified model cannot accurately predict the lock contention. For *Volrend* and *Radiosity*, the model over simplifies that lock access arrives uniformly within the parallel section, thus under estimate the lock contention. For *Canneal*, it assume a constant critical section size, however, the critical section length decreases due to each thread given less work when scaling the number of threads.

5.3.6 Model application

In addition to using our model as a performance prediction tool, it can also be used as a performance debugging tool. Since our model predicts the lock contention for each lockPC, we can easily identify which lockPC is contributing to the most lock contention. In addition, because our model breaks down the lock contention into inter-arrival rate, critical section length, lock access histogram, and lock handoff latency, one can use such information to identify which component is the main culprit for the increase in lock contention.

For example, in *Volrend*, there are 3 lockPCs, with 1 lockPC contributing to 95% of lock contention. After examining the model parameters, we were able to identify that the lock contention is not caused by large and frequent amount of lock access, but instead is due to large critical section length. Thus, we were able to identify the main contributor in lock contention, and suggest that in order to reduce such lock contention, one would need to reduce the work done in the critical section

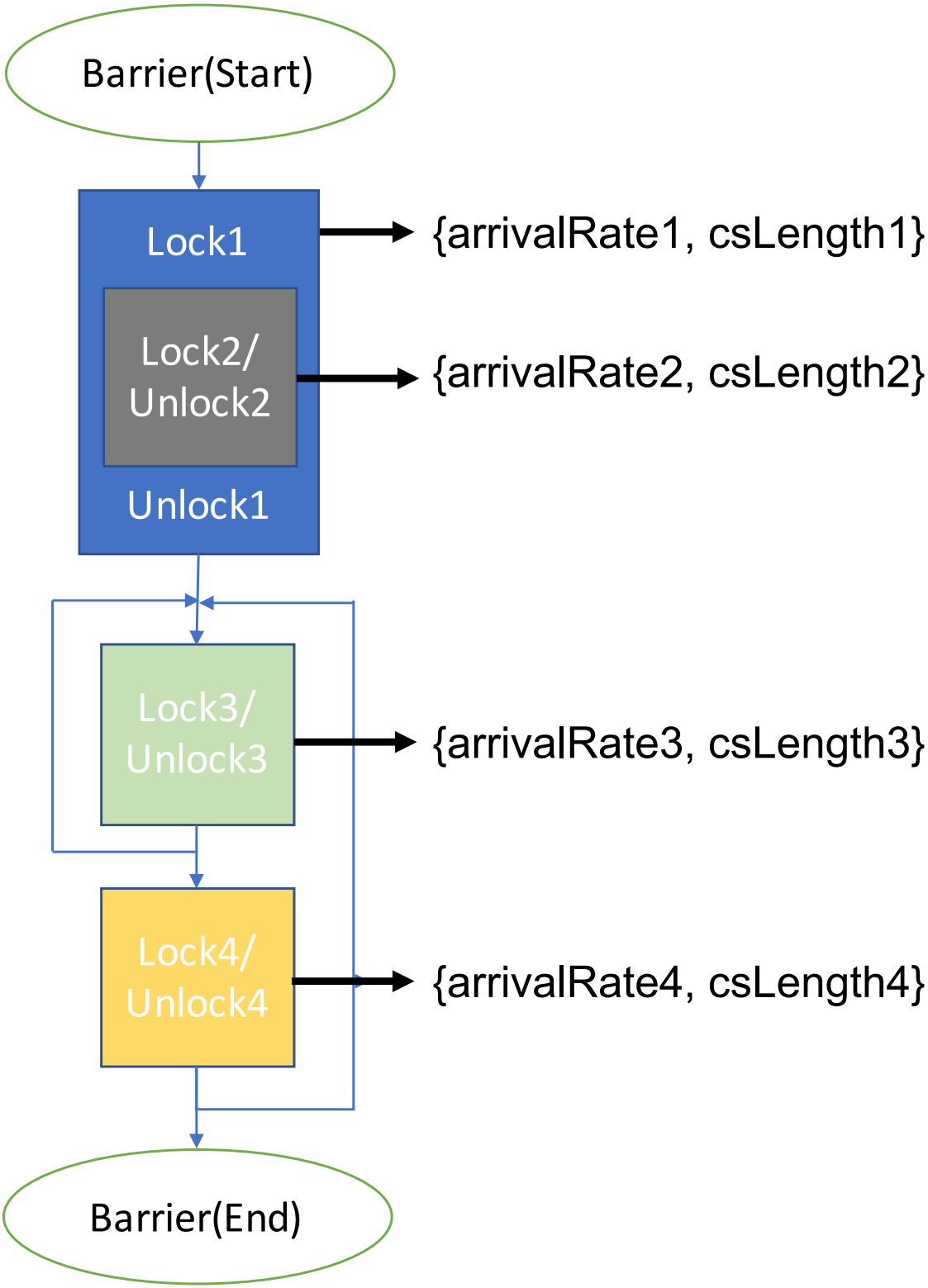

Figure 5.1: Parallel section/LockPC prediction scheme

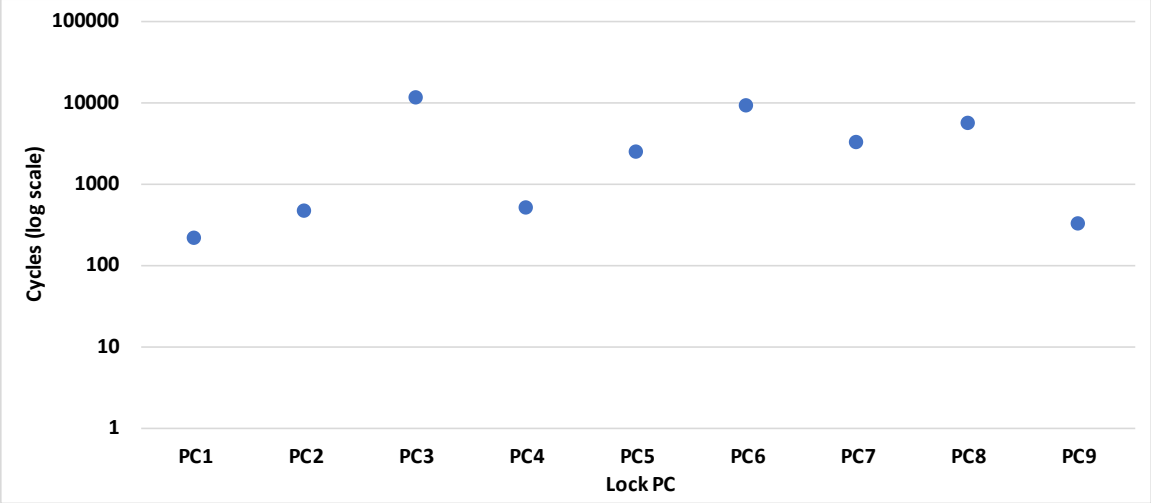

Figure 5.2: Average inter-arrival time for different LockPC

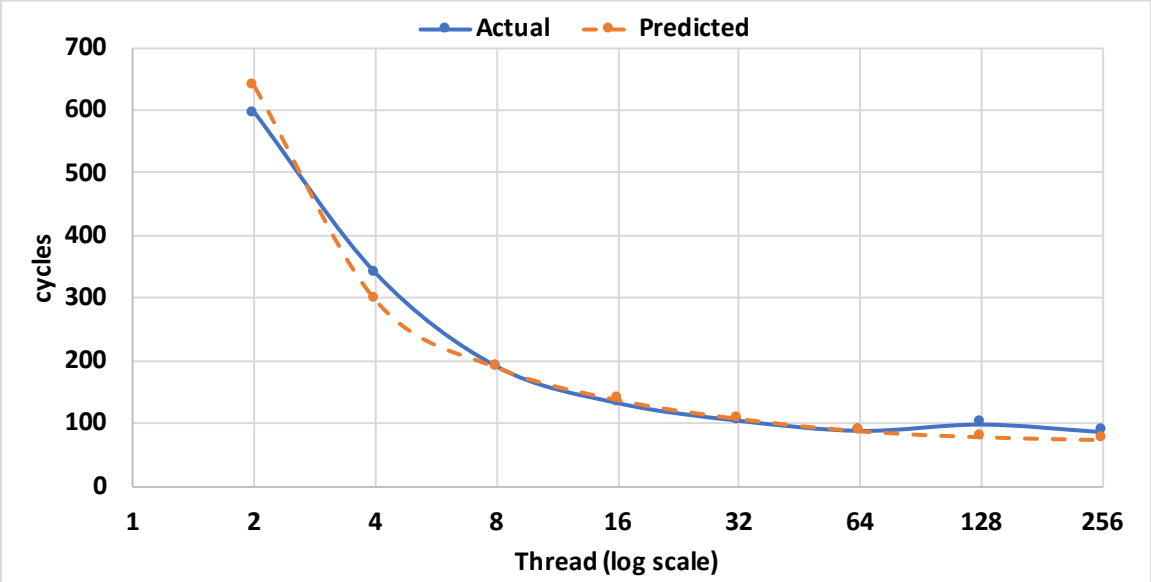

Figure 5.3: Predicting average inter-arrival time using regression model

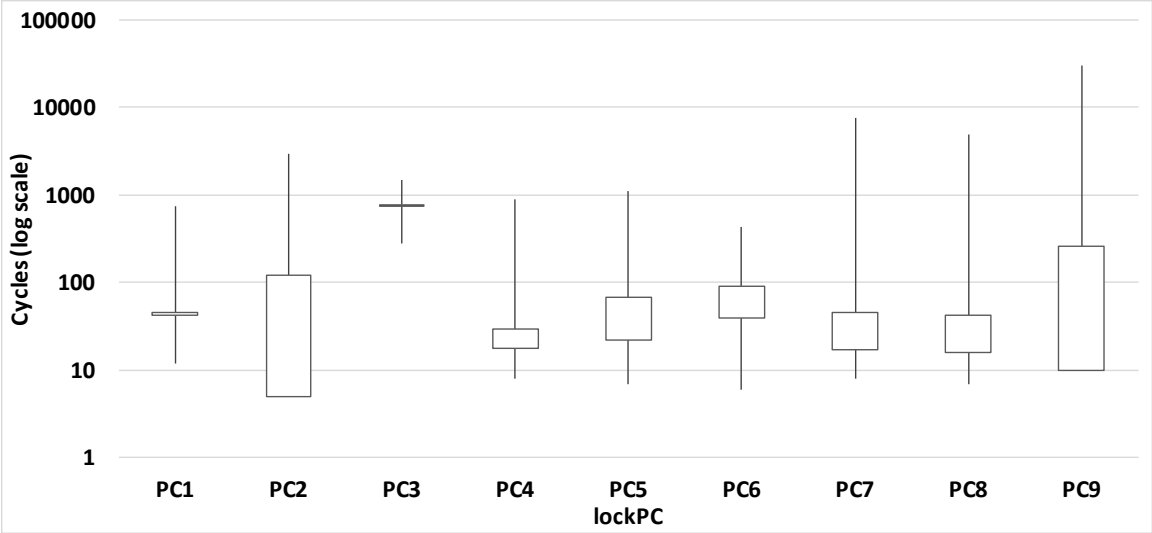

Figure 5.4: Critical section size for different lockPC

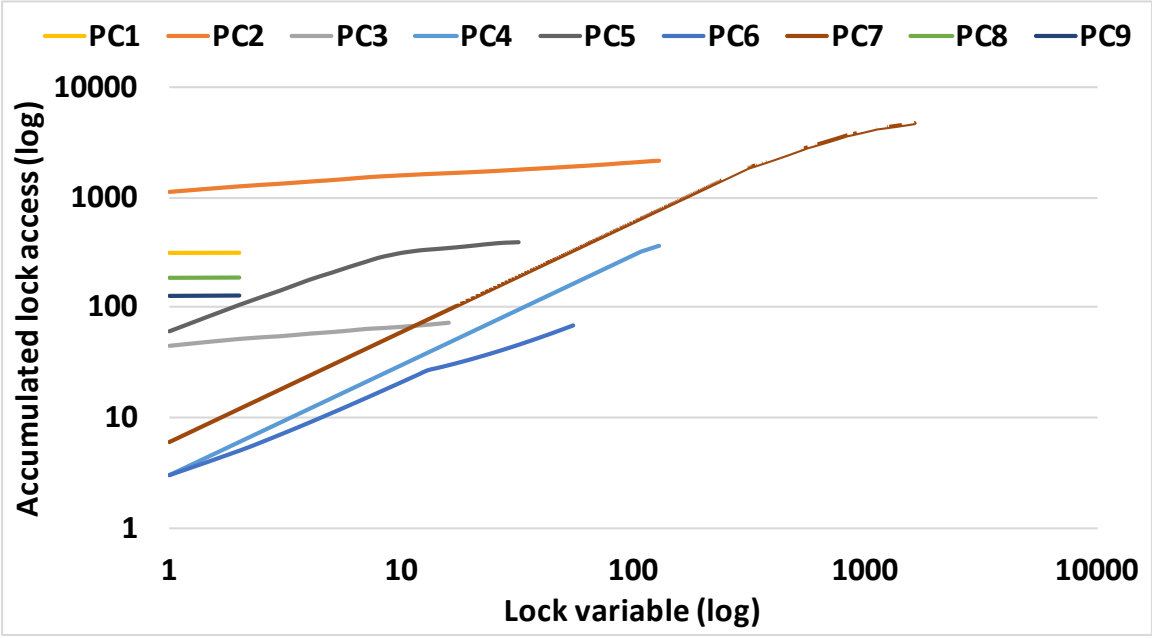

Figure 5.5: Lock access histogram for different lockPC

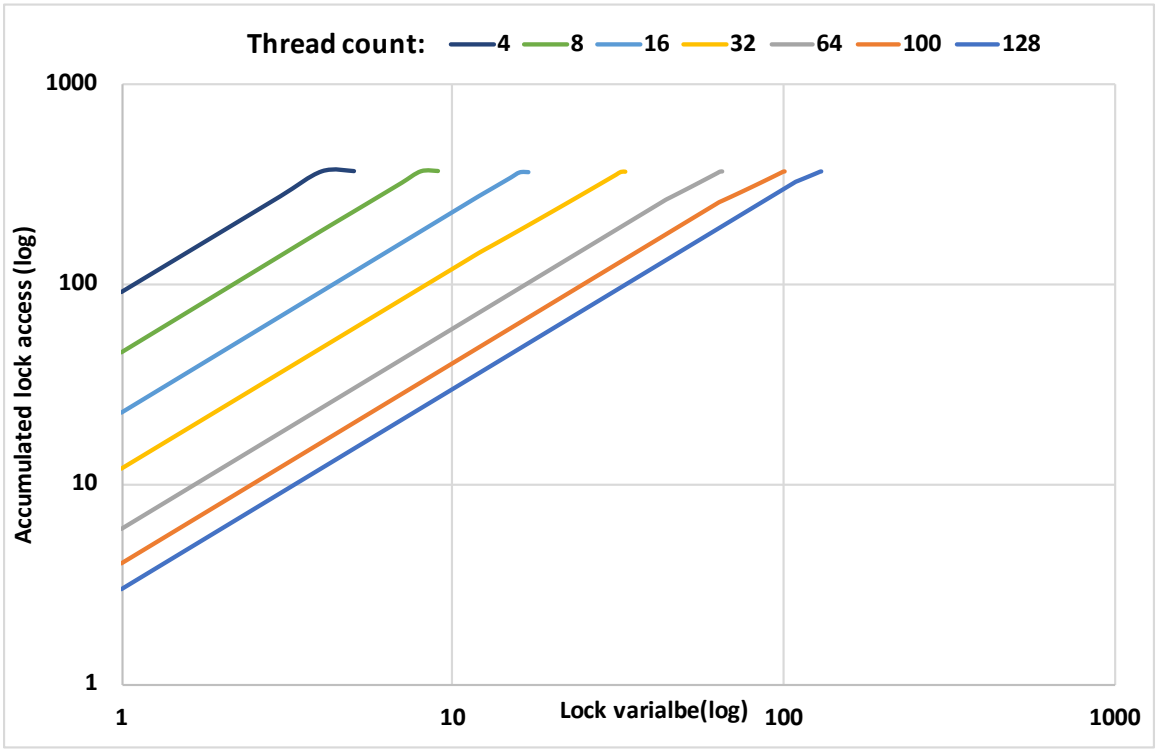

Figure 5.6: Prediction of lock access histogram with varying thread count

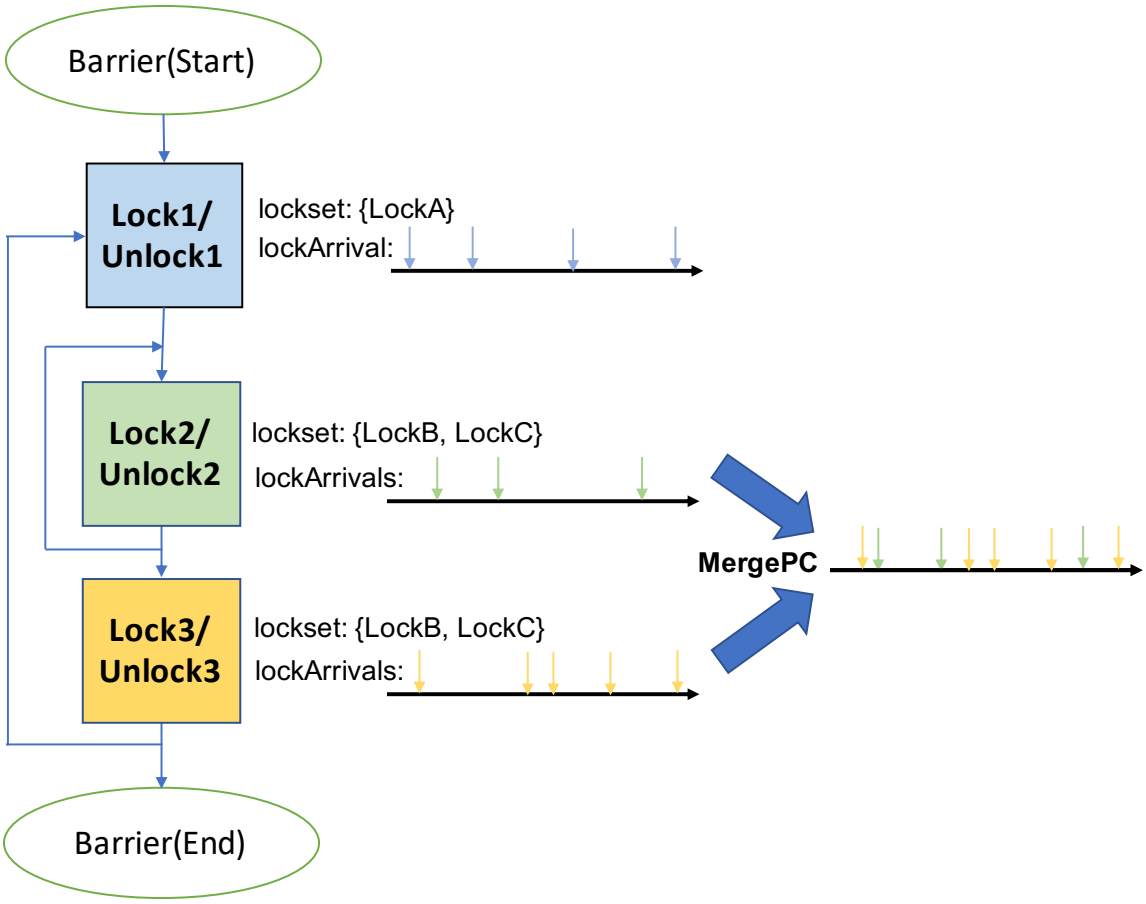

Figure 5.7: Merging statistics for various lockPCs

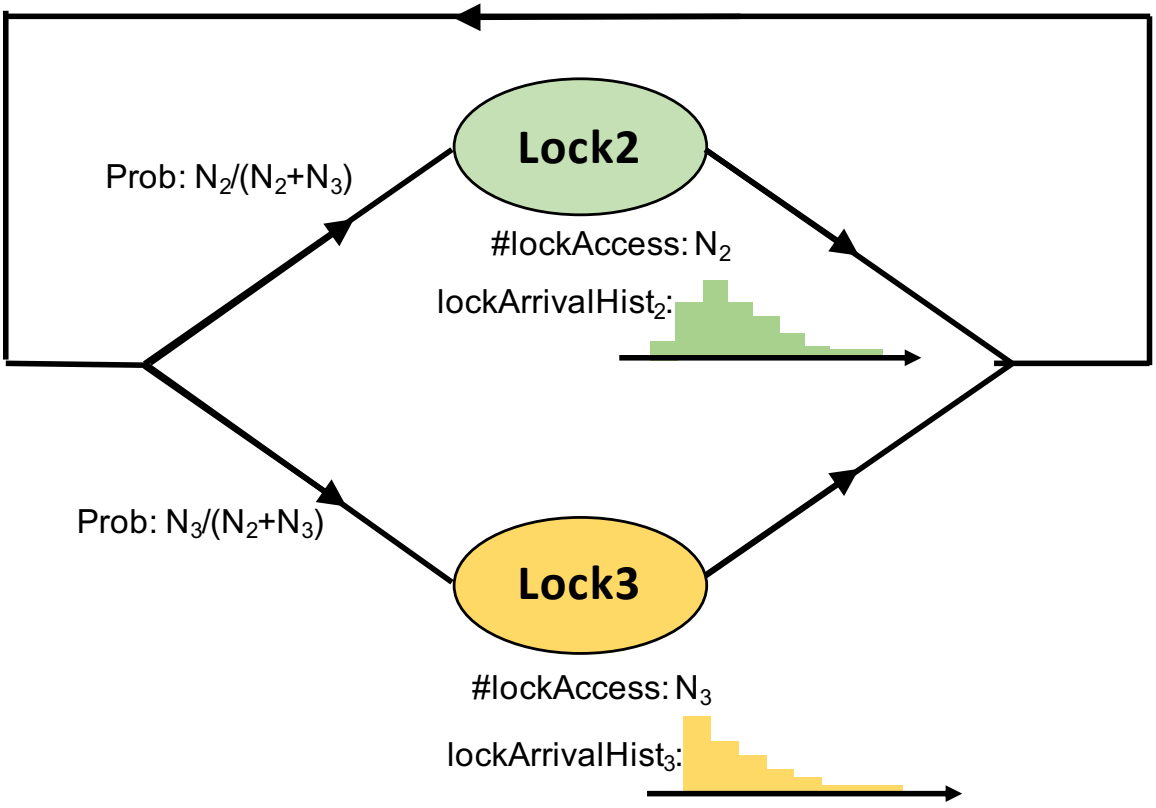

Figure 5.8: Evaluate merged lockPCs lock contention

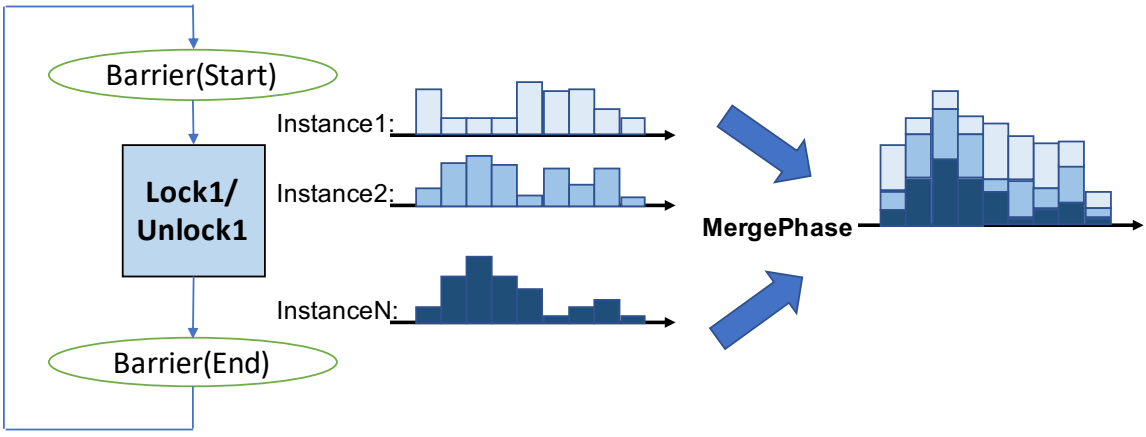

Figure 5.9: Merging statistics for different parallel sections

Table 5.2: Summary of benchmark characteristic on lock accesses

Benchmark	**Unique barrier phases**	Lock-per-million-instruction (LPMI)	**Lock cotention**	**Reason**
Barnes	4	28	7%	Short inter-arrival time
Cholesky	5	81	18%	Large critical section
Ocean	8	112	17%	Synchronious arrival of threads
Radiosity	4	167	34%	Short inter-arrival time and large handoff latency
Raytrace	2	179	71%	Short inter-arrival time and large critical section
Volrend	5	62	38%	Large critical section
Canneal	7	17	48%	Short inter-arrival time and large critical section
Fluidanimate	10	4344	0%	Fine-grain locking

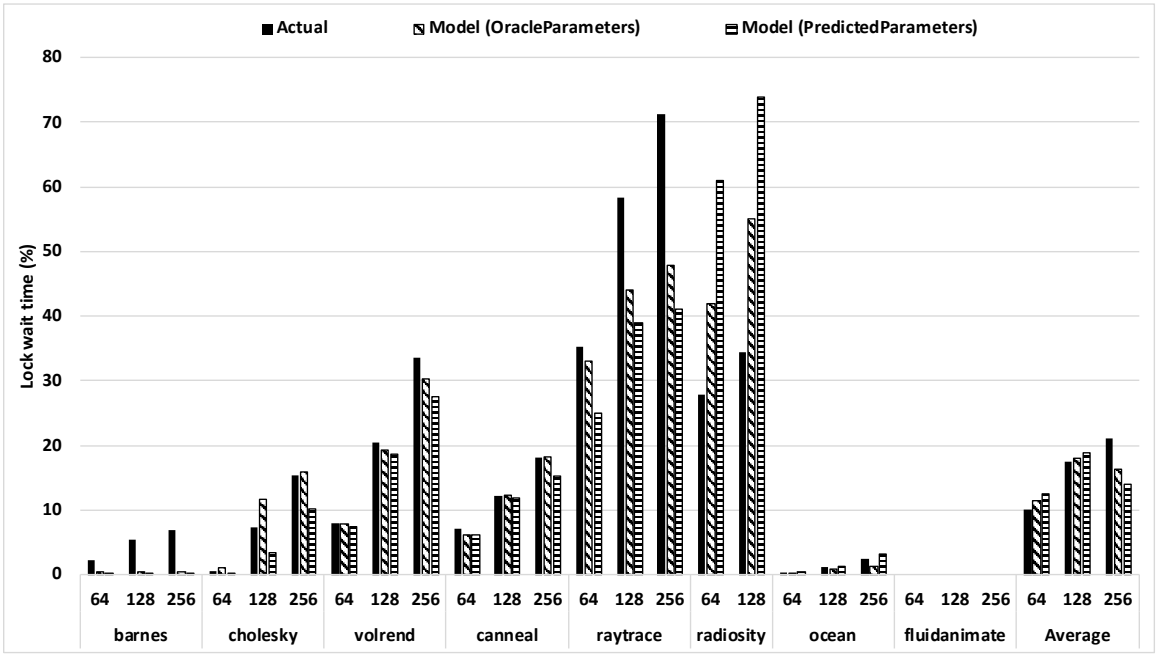

Figure 5.10: Average per-thread lock wait time over total runtime

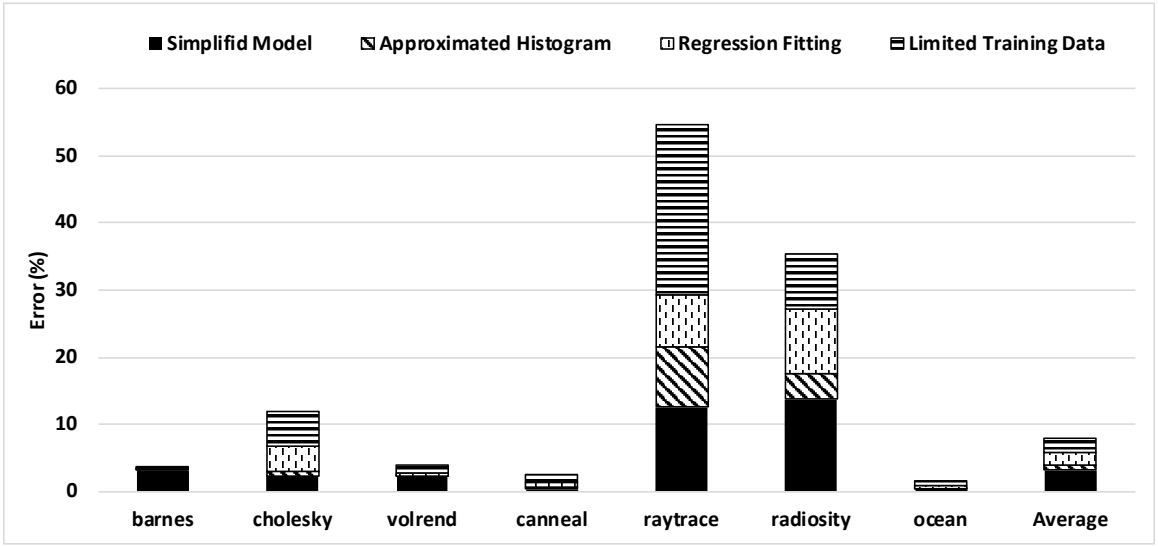

Figure 5.11: Average peedup prediction error

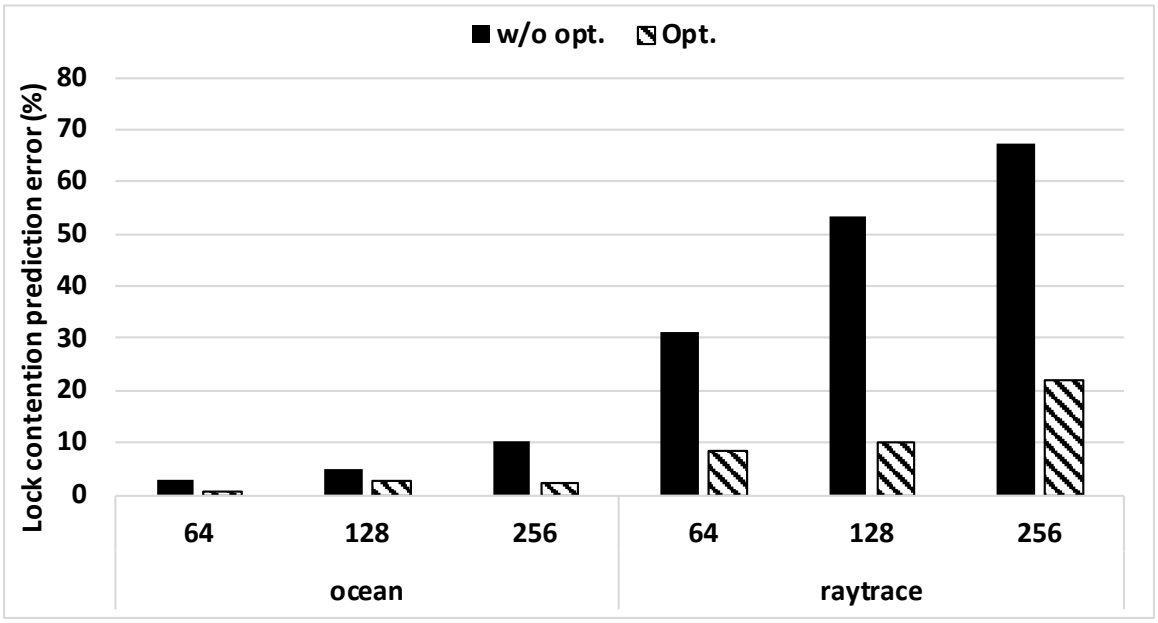

Figure 5.12: Effectiveness of optimization on lock contention prediction accuracy

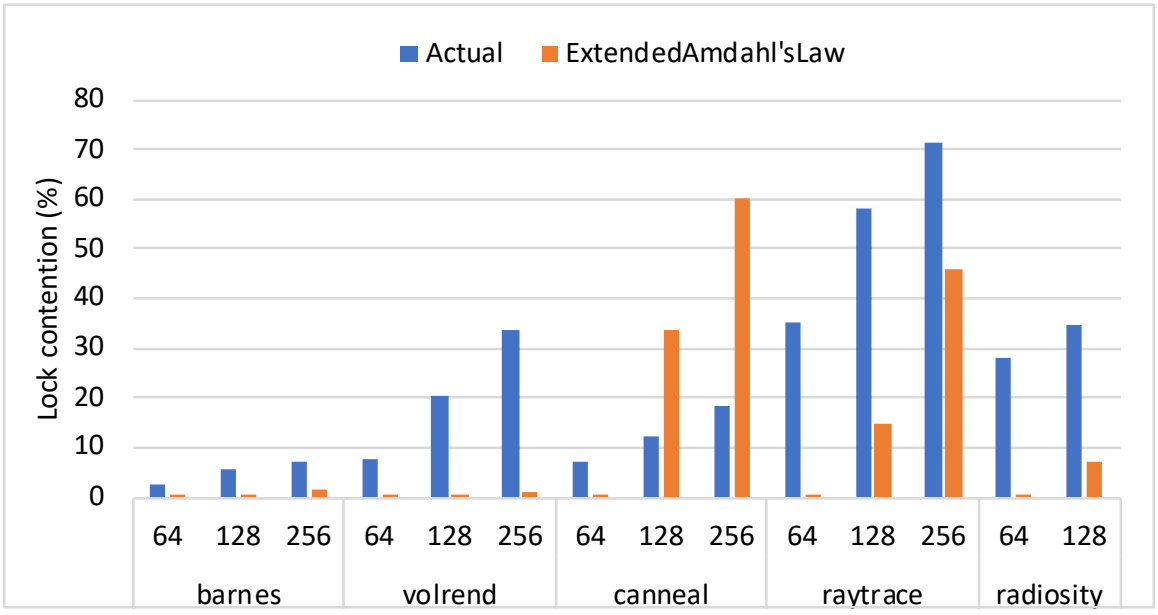

Figure 5.13: Lock contention prediction result using extended Amdahl's law

CHAPTER 6

CONCLUSION

In recent years, the number of available cores in a processor is increasing rapidly while the pace of performance improvement of an individual core has been lagged. As a result, applications are now required to extract more parallelism and leverage the abundant number of cores to ensure continuous speedup of their applications. However, ensure application scale well over many threads is a challenge task, mainly because scalability bottlenecks such as synchronization will saturate the performance gain if not managed carefully. In addition, finding the optimal thread count to balance the overhead and benefit of penalization becomes even more critical.

In Chapter 3, I presented *MiSAR*, a minimalistic synchronization accelerator (MSA) that supports the three commonly used synchronizations, along with a small and efficient overflow management unit (OMU) that safely and dynamically manages the MSA's limited hardware resources. Our results indicate that in a 64-core processor, the OMU allows a 2-entry-per-tile MSA to service 93% of synchronization operations on average, achieving an average speedup of 1.43X (up to 7.59X in streamcluster!) over the software (pthreads) implementation, and performing within 3% of ideal (zero-latency) synchronization.

In Chapter 4, I presented a new performance model that captures program characteristics of multi-threaded applications, allowing it to use few-threaded runs along with small input sets to predict performance of many-threaded runs with large input sets. First, we partition the program execution into barrier phases, and model the scaling trend of the total instruction count and its distribution among threads for each barrier phase in order to account for parallelization overheads. Second, we subdivide each barrier phase into small intervals, and model the cache miss rate of each interval by utilizing the regular shifting of concurrent reuse distance (CRD) profiles. Applying the CRD analysis to small inter-

vals allows the CRD profile to capture behavior and model performance of each phase of the program individually, rather than trying to model the aggregate behavior of potentially many phases that may differ widely in terms of cache capacity and memory bandwidth demand. Third, we use a simplified DRAM model to capture the impact of the memory subsystem on the total execution time. Finally, we model how the number of barrier phases and the model parameters (instruction count and CRD) changes with input size to predict across different input sets. Overall, our model has only a 27% error when predicting parallel speedup for 32- to 256-core runs when model parameters are extracted from 1- to 16-core runs. Our model's prediction of the performance-optimal number of threads for an application is within 40% or the actual optimum, compared to a 200% error when using a simple model based on the extended Amdahl's Law.

In Chapter 5, I presented a new PC-based lock contention model that leverage the structural change in program characteristics to predict the lock contention by modeling how the lock arrival rate, the critical section, and also the lockPC-to-lockAddress mapping changes under thread-scaling and input-scaling. Our lock contention model consists of 4 parts. First, we divide the program execution into parallel phases separated by global synchronization (barrier, fork-join, etc.). Second, we collect statistics that represent the synchronicity of thread arrival (lock arrival rate) as well as the functionality of the corresponding critical section (size of the critical section) for each lock PC. Third, we approximate the rates into well-known statistic models (eq. exponential distribution, gaussian distribution, etc.) in order to reduce the parameters required to model the lock contention. Last, we use regression models to predict how the parameters will change when varying the number of locks and input size. Overall, our model was able to predict within 7% of lock contention when using oracle parameters for our model, and an additional 6% error using predicted parameters using training data from 1- to 32-thread runs. This shows the effectiveness of our PC-based lock contention model and enables application developers to better understand and predict how the application's lock contention will scale when increasing thread count.

The trend of increasing amount of processing cores will emphasize the importance of modeling, predicting, and mitigating the scalability bottlenecks. By using statistical modeling techniques, we can better identify and predict any scalability bottleneck that may occur and react accordingly. By using hardware accelerators, we can mitigate the severity of the synchronization bottleneck, thus allowing the application to perform better with increasing number of threads. This dissertation serves as a starting point to investigate techniques to better model, predict, and mitigate the scaling of parallel applications in order to fully utilize the abundant amount of processing cores on future processors.

REFERENCES

[1] C. J. Beckmann and C. D. Polychronopoulos, "Fast barrier synchronization hardware," in *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '90, New York, New York, United States: IEEE Computer Society Press, 1990, pp. 180–189, ISBN: 0-89791-412-0.

[2] A. Kägi, D. Burger, and J. R. Goodman, "Efficient synchronization: Let them eat qolb," in *Proceedings of the 24th annual international symposium on Computer architecture*, ser. ISCA '97, Denver, Colorado, United States: ACM, 1997, pp. 170–180, ISBN: 0-89791-901-7.

[3] J. T. Robinson, "A fast general-purpose hardware synchronization mechanism," in *Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '85, Austin, Texas, United States: ACM, 1985, pp. 122–130, ISBN: 0-89791-160-1.

[4] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, "The mit alewife machine: Architecture and performance," in *Proceedings of the 22nd annual international symposium on Computer architecture*, ser. ISCA '95, S. Margherita Ligure, Italy: ACM, 1995, pp. 2–13, ISBN: 0-89791-698-0.

[5] G. Almási, C. Archer, J. G. Castaños, J. A. Gunnels, C. C. Erway, P. Heidelberger, X. Martorell, J. E. Moreira, K. Pinnow, J. Ratterman, B. D. Steinmacher-Burow, W. Gropp, and B. Toonen, "Design and implementation of message-passing services for the blue gene/l supercomputer," *IBM Journal of Research and Development*, vol. 49, no. 2.3, pp. 393 –406, 2005.

[6] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The tera computer system," in *Proceedings of the 4th international conference on Supercomputing*, ser. ICS '90, Amsterdam, The Netherlands: ACM, 1990, pp. 1–6, ISBN: 0-89791-369-8.

[7] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The nyu ultracomputerdesigning a mimd, shared-memory parallel machine (extended abstract)," in *Proceedings of the 9th annual symposium on Computer Architecture*, ser. ISCA '82, Austin, Texas, United States: IEEE Computer Society Press, 1982, pp. 27–42.

[8] J. Laudon and D. Lenoski, "The sgi origin: A ccnuma highly scalable server," in *Proceedings of the 24th annual international symposium on Computer architecture*,

ser. ISCA '97, Denver, Colorado, United States: ACM, 1997, pp. 241–251, ISBN: 0-89791-901-7.

[9] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak, "The network architecture of the connection machine cm-5 (extended abstract)," in *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '92, San Diego, California, United States: ACM, 1992, pp. 272–285, ISBN: 0-89791-483-X.

[10] S. L. Scott, "Synchronization and communication in the t3e multiprocessor," in *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-VII, Cambridge, Massachusetts, United States: ACM, 1996, pp. 26–36, ISBN: 0-89791-767-7.

[11] J. Abellán, J. Fernández, and M. Acacio, "A g-line-based network for fast and efficient barrier synchronization in many-core cmps," in *Parallel Processing (ICPP), 2010 39th International Conference on*, 2010, pp. 267–276.

[12] ——, "Glocks: Efficient support for highly-contended locks in many-core cmps," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, 2011, pp. 893 –905.

[13] J. Oh, M. Prvulovic, and A. Zajic, "Tlsync: Support for multiple fast barriers using on-chip transmission lines," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, 2011, pp. 105 –115.

[14] E. Vallejo, R. Beivide, A. Cristal, T. Harris, F. Vallejo, O. Unsal, and M. Valero, "Architectural support for fair reader-writer locking," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43, Washington, DC, USA: IEEE Computer Society, 2010, pp. 275–286, ISBN: 978-0-7695-4299-7.

[15] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao, "Synchronization state buffer: Supporting efficient fine-grain synchronization on many-core architectures," in *Proceedings of the 34th annual international symposium on Computer architecture*, ser. ISCA '07, San Diego, California, USA: ACM, 2007, pp. 35–45, ISBN: 978-1-59593-706-3.

[16] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu, "Knights landing: Second-generation intel xeon phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

[17] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C. C. Miao, C. Ramey, D. Wentzlaff, W. Anderson,

E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "Tile64 - processor: A 64-core soc with mesh interconnect," in *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, 2008, pp. 88–598.

[18] W. Heirman, T. E. Carlson, K. V. Craeynest, I. Hur, A. Jaleel, and L. Eeckhout, "Undersubscribed threading on clustered cache architectures," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 678–689.

[19] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, "Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on cmps," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII, Seattle, WA, USA: ACM, 2008, pp. 277–286, ISBN: 978-1-59593-958-6.

[20] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII, San Jose, California, USA: ACM, 2006, pp. 185–194, ISBN: 1-59593-451-0.

[21] X. Liu and B. Wu, "Scaanalyzer: A tool to identify memory scalability bottlenecks in parallel programs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15, Austin, Texas: ACM, 2015, 47:1–47:12, ISBN: 978-1-4503-3723-6.

[22] S. Eyerman, K. D. Bois, and L. Eeckhout, "Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications," in *2012 IEEE International Symposium on Performance Analysis of Systems Software*, 2012, pp. 145–155.

[23] M. Kim, P. Kumar, H. Kim, and B. Brett, "Predicting potential speedup of serial code via lightweight profiling and emulations with memory performance model," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, ser. IPDPS '12, Washington, DC, USA: IEEE Computer Society, 2012, pp. 1318–1329, ISBN: 978-0-7695-4675-9.

[24] S. Demetriades and S. Cho, "Barrierwatch: Characterizing multithreaded workloads across and within program-defined epochs," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, ser. CF '11, Ischia, Italy: ACM, 2011, 5:1–5:11, ISBN: 978-1-4503-0698-0.

[25] T. E. Carlson, W. Heirman, K. V. Craeynest, and L. Eeckhout, "Barrierpoint: Sampled simulation of multi-threaded applications," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 2–12.

[26] M. J. Wu and D. Yeung, "Coherent profiles: Enabling efficient reuse distance analysis of multicore scaling for loop-based parallel programs," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 264–275.

[27] M.-J. Wu, M. Zhao, and D. Yeung, "Studying multicore processor scaling via reuse distance analysis," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13, Tel-Aviv, Israel: ACM, 2013, pp. 499–510, ISBN: 978-1-4503-2079-5.

[28] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *Proceedings of the 22nd annual international symposium on Computer architecture*, ser. ISCA '95, S. Margherita Ligure, Italy: ACM, 1995, pp. 24–36, ISBN: 0-89791-698-0.

[29] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *11th International Symposium on High-Performance Computer Architecture*, 2005, pp. 340–351.

[30] C. Ding and T. Chilimbi, "A composable model for analyzing locality of multi-threaded programs," Tech. Rep., 2009.

[31] Mediatek. (). Mediatek helio x30.

[32] C.-K. Liang and M. Prvulovic, "Misar: Minimalistic synchronization accelerator with resource overflow management," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15, Portland, Oregon: ACM, 2015, pp. 414–426, ISBN: 978-1-4503-3402-0.

[33] R. Rajwar and J. R. Goodman, "Speculative lock elision: Enabling highly concurrent multithreaded execution," in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 34, Austin, Texas: IEEE Computer Society, 2001, pp. 294–305, ISBN: 0-7695-1369-7.

[34] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, "Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on cmps," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII, Seattle, WA, USA: ACM, 2008, pp. 277–286, ISBN: 978-1-59593-958-6.

[35] S. Sridharan, G. Gupta, and G. S. Sohi, "Adaptive, efficient, parallel execution of parallel programs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, Edinburgh, United Kingdom: ACM, 2014, pp. 169–180, ISBN: 978-1-4503-2784-8.

[36] Y. Cui, Y. Wang, Y. Chen, and Y. Shi, "Lock-contention-aware scheduler: A scalable and energy-efficient method for addressing scalability collapse on multicore systems," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, 44:1–44:25, Jan. 2013.

[37] G. Chen and P. Stenstrom, "Critical lock analysis: Diagnosing critical section bottlenecks in multithreaded applications," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, Salt Lake City, Utah: IEEE Computer Society Press, 2012, 71:1–71:11, ISBN: 978-1-4673-0804-5.

[38] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout, "Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13, Tel-Aviv, Israel: ACM, 2013, pp. 511–522, ISBN: 978-1-4503-2079-5.

[39] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield, "Analyzing lock contention in multithreaded applications," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '10, Bangalore, India: ACM, 2010, pp. 269–280, ISBN: 978-1-60558-877-3.

[40] M. M. u. Alam, T. Liu, G. Zeng, and A. Muzahid, "Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17, Belgrade, Serbia: ACM, 2017, pp. 298–313, ISBN: 978-1-4503-4938-3.

[41] Intel, *Intel VTune Performance Analyzer*, http://www.intel.com/cd/software/products/asmona/eng/vtune/239144.htm, 2008.

[42] M. V. Studio. (). Concurrency visualize.

[43] P. S. Yu, D. M. Dias, and S. S. Lavenberg, "On the analytical modeling of database concurrency control," *J. ACM*, vol. 40, no. 4, pp. 831–872, Sep. 1993.

[44] A. Thomasian, "On a more realistic lock contention model and its analysis," in *Proceedings of 1994 IEEE 10th International Conference on Data Engineering*, 1994, pp. 2–9.

[45] S. Eyerman and L. Eeckhout, "Modeling critical sections in amdahl's law and its implications for multicore design," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10, Saint-Malo, France: ACM, 2010, pp. 362–370, ISBN: 978-1-4503-0053-7.

[46] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, "Non-scalable locks are dangerous," 2012.

[47] S. Keckler, W. Dally, D. Maskit, N. Carter, A. Chang, and W. Lee, "Exploiting fine-grain thread level parallelism on the mit multi-alu processor," in *Computer Architecture, 1998. Proceedings. The 25th Annual International Symposium on*, 1998, pp. 306 –317.

[48] J. Sampson, R. González, J.-F. Collard, N. P. Jouppi, M. Schlansker, and B. Calder, "Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39, Washington, DC, USA: IEEE Computer Society, 2006, pp. 235–246, ISBN: 0-7695-2732-9.

[49] L. Zhang, Z. Fang, and J. Carter, "Highly efficient synchronization based on active memory operations," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004, p. 58.

[50] M.-C. Chiang, "Memory system design for bus-based multiprocessors," UMI Order No. GAX92-09300, PhD thesis, Madison, WI, USA, 1992.

[51] B. S. Akgul, J. Lee, and V. J. Mooney, "A system-on-a-chip lock cache with task preemption support," in *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, ser. CASES '01, Atlanta, Georgia, USA: ACM, 2001, pp. 149–157, ISBN: 1-58113-399-5.

[52] F. Petrini, J. Fernandez, E. Frachtenberg, and S. Coll, "Scalable collective communication on the asci q machine," in *High Performance Interconnects, 2003. Proceedings. 11th Symposium on*, 2003, pp. 54 –59.

[53] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz, "A regression-based approach to scalability prediction," in *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ser. ICS '08, Island of Kos, Greece: ACM, 2008, pp. 368–377, ISBN: 978-1-60558-158-3.

[54] E. Ïpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, "Efficiently exploring architectural design spaces via predictive modeling," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII, San Jose, California, USA: ACM, 2006, pp. 195–206, ISBN: 1-59593-451-0.

[55] W. Wang, J. W. Davidson, and M. L. Soffa, "Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale numa machines," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 419–431.

[56] W. Wang, T. Dey, J. W. Davidson, and M. L. Soffa, "Dramon: Predicting memory bandwidth usage of multi-threaded programs with high accuracy and low overhead,"

in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 380–391.

[57] N. Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan, "Anatomy: An analytical model of memory system performance," in *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '14, Austin, Texas, USA: ACM, 2014, pp. 505–517, ISBN: 978-1-4503-2789-3.

[58] G. E. Suh, S. Devadas, and L. Rudolph, "Analytical cache models with applications to cache partitioning," in *ACM International Conference on Supercomputing 25th Anniversary Volume*, Munich, Germany: ACM, 2014, pp. 323–334, ISBN: 978-1-4503-2840-1.

[59] W. Heirman, T. E. Carlson, S. Che, K. Skadron, and L. Eeckhout, "Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, 2011, pp. 38–49.

[60] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs http://hpctoolkit.org," *Concurr. Comput. : Pract. Exper.*, vol. 22, no. 6, pp. 685–701, Apr. 2010.

[61] Y. He, C. E. Leiserson, and W. M. Leiserson, "The cilkview scalability analyzer," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10, Thira, Santorini, Greece: ACM, 2010, pp. 145–156, ISBN: 978-1-4503-0079-7.

[62] Y. Xiao, H. Zhengyu, and H. Bo, "A queuing modelbased approach for the analysis of transactional memory systems," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 6, pp. 808–825,

[63] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, *Sesc simulator, january 2005*.

[64] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2003.

[65] C. Bienia, "Benchmarking modern multiprocessors," PhD thesis, Princeton University, 2011.

[66] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, Feb. 1991.

[67] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05, Chicago, IL, USA: ACM, 2005, pp. 190–200, ISBN: 1-59593-056-6.

[68] J. S. Harper, D. J. Kerbyson, and G. R. Nudd, "Analytical modeling of set-associative cache behavior," *IEEE Transactions on Computers*, vol. 48, no. 10, pp. 1009–1024, 1999.

[69] C.-K. Liang and M. Prvulovic, "Misar: Minimalistic synchronization accelerator with resource overflow management," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15, Portland, Oregon: ACM, 2015, pp. 414–426, ISBN: 978-1-4503-3402-0.

[70] J. Oh, M. Prvulovic, and A. Zajic, "Tlsync: Support for multiple fast barriers using on-chip transmission lines," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11, San Jose, California, USA: ACM, 2011, pp. 105–116, ISBN: 978-1-4503-0472-6.

[71] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Comput. Archit. Lett.*, vol. 10, no. 1, pp. 16–19, Jan. 2011.

[72] S. Sridharan, G. Gupta, and G. S. Sohi, "Adaptive, efficient, parallel execution of parallel programs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, Edinburgh, United Kingdom: ACM, 2014, pp. 169–180, ISBN: 978-1-4503-2784-8.