EFFICIENT HARDWARE AND SOFTWARE ASSIST FOR MANY-CORE PERFORMANCE

A Thesis
Presented to
The Academic Faculty

by

Jungju Oh

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

School of Computer Science
Georgia Institute of Technology
December 2013

EFFICIENT HARDWARE AND SOFTWARE ASSIST FOR MANY-CORE PERFORMANCE

Approved by:

Professor Milos Prvulovic, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Alenka Zajic, Co-advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Tom Conte
School of Computer Science
Georgia Institute of Technology

Professor Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Professor Moinuddin K. Qureshi
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: 22 July 2013

To my beloved wife and our first son to be born

from whom I have received endless support,

and to my parents and brother whom I aspire to live and love like

PREFACE

This thesis is about how we can do better in the many-core era. It begins with how we can run applications faster with many-core processors, even with the heterogeneous ones. It continues with how we can synchronize faster with more cores. It ends with how we can communicate faster between cores drifting apart in many-core processors. Essentially, this thesis provides vision for the upcoming obstacles with the ever increasing number of cores.

ACKNOWLEDGEMENTS

I would like to express the deepest appreciation to my advisors, Dr. Milos Prvulovic and Dr. Alenka Zajic. They patiently provided the vision, encouragement and advise necessary for me to see the light at the end of the tunnel. And I also thank my kind committee members, Dr. Tom Conte, Dr. Hyesoon Kim, and Dr. Moinuddin Qureshi for their support and guidance. I extend my gratitude to my internship mentor at Intel, Dr. Christopher J. Hughes for his support, guidance and helpful suggestions.

My friends were sources of laughter and joy. They also gave me emotional support to never give up on this long, winding path. I thank to my lab mates, Guru, Ioannis, Anshuman, Kyle, and Sunjae. I am particularly grateful for the assistance given by my lunch mates, Minsung, Moonkyung, Sangmin, and Hyojoon.

I wish to thank my parents, Seungche Oh and Keumrye Lee. Their love was my driving force. I owe them everything and wish I could show how much I love them.

Last but not the least, I owe my deepest gratitude to my beloved wife who cheered me up and supported me everyday, for which reason I am forever indebted and do not know how to pay back.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Software frameworks and hardware techniques that address scalability limiters can cooperatively improve the performance of parallel applications on many-core processors.

In recent years, the number of available cores in a processor are increasing rapidly while the pace of performance improvement of an individual core has been lagged. It led application developers to extract more parallelism from a number of cores to make their applications run faster. However, writing a parallel program that scales well with the increasing core counts is challenging. Consequently, many parallel applications suffer from *performance bugs* caused by *scalability limiters*.

We expect core counts to continue to increase for the foreseeable future and hence, addressing scalability limiters is important for better performance on future hardware. With this thesis, I propose both software frameworks and hardware improvements that I developed to address three important scalability limiters: load imbalance, barrier latency and increasing on-chip packet latency.

First, I introduce a debugging framework for load imbalance called LIME. The LIME framework uses profiling, statistical analysis and control flow graph analysis to automatically determine the nature of load imbalance problems and pinpoint the code where the problems are introduced.

Second, I address scalability problem of the barrier, which has become costly and difficult to achieve scalable performance. To address this problem, I propose a transmission line (TL) based hardware barrier support, called TLSync, that is orders of magnitude faster than software barrier implementation while supports many (tens)

of barriers simultaneously using a single chip-spanning network.

Third and lastly, I focus on the increasing packet latency in on-chip network, and propose a hybrid interconnection where a low-latency TL based interconnect is synergistically used with a high-throughput switched interconnect. Also, a new adaptive packet steering policy is created to judiciously use the limited throughput available on the low-latency TL interconnect.

CHAPTER I

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, application developers are now required to extract more parallelism and leverage the abundant number of cores to make their applications run faster. In the single-core era, the improvement in core performance is translated into a speedup of an application without any change on the application. However, the increasing core counts barely contribute to performance gain of a parallel application unless written specifically to *scale*, and writing a scalable parallel program is a challenging task, putting aside the problem of writing a correct program. Many parallel applications have suffered from *performance bugs* caused by *scalability limiters*. These prevent performance from improving as much as it should with more cores.

Since we expect core counts to continue to grow for the foreseeable future, addressing scalability limiters is crucial for software development in order to obtain better performance on future hardware. However, managing scalability limiters is not a straightforward task because numerous factors can lead to various types of scalability limiters that are either software- or hardware-related ones, or both. For example, a program may not scale with more cores because the algorithm implemented in the program is unable to offload proper workload to the additional cores, causing software-related scalability limiters [72]. It is also possible that the application interacts with underlying hardware in a biased way, causing an unbalanced progress between cores. This may incur stall at synchronization point and deteriorates scalability. In that case, the scalability limiter is related to both hardware and software.

Thus, in this work, I propose both a software framework and hardware improvements that I developed to identify and manage important scalability limiters. Specifically, I target three limiters: load imbalance, barrier latency and increasing on-chip packet latency.

1.2 Scalability Limiters

1.2.1 Load Imbalance and Debugging Framework

Load imbalance is one of the key scalability limiters in parallel applications. Ideally, a parallel application assigns an equal amount of work to all available cores, keeping all of them busy for the parallel section delimited by global synchronization points (i.e., barriers). Load imbalance occurs when some cores run out of work and must wait for the remaining cores to finish their work.

One characteristic of load imbalance is that is relatively easy to detect—one can watch for threads waiting at the end of a parallel section or at a thread-join point, but it is much more difficult to diagnose the cause of load imbalance. In particular, to help programmers decide what changes to make to the application to resolve the imbalance, the cause of load imbalance should be identified with sufficient precision. However, it may be difficult to diagnose the cause of load imbalance because of the wide range of candidates. For instance, load imbalance can be caused by assigning an unfair proportion of tasks to a thread, or by assigning too many long tasks to the same thread—both of these manifest as control flow differences between threads. Diagnosing causes for imbalance becomes even harder when it occurs due to interactions between the application and the underlying hardware (e.g., threads having different numbers of cache misses). Such causes cannot be easily detected through code inspection or static analysis.

Reducing load imbalance has long been an active research topic. The most common approach to reducing load imbalance is to use customized software stack that supports dynamic task scheduling, such as OpenMP [75] and TBB [42]. Dynamic task scheduling involves partitioning the parallel work into many more tasks than threads, and using a run-time system to assign tasks to threads on-demand. Dynamic scheduling significantly reduces load imbalance, but introduces considerable runtime overheads, both from executing scheduling code and from the loss of cache locality among tasks. These overheads increase with the number of cores/threads, and can dominate performance [57].

Because load imbalance is a major issue in debugging the performance of parallel programs, and because existing approaches do not satisfactorily address this problem, there is a need for tools to help programmers efficiently find what causes load imbalance in their code. Therefore, I propose a framework for debugging load imbalance, called LIME. This framework uses profiling, statistical analysis, and control flow graph analysis to automatically determine the nature of load imbalance problems and pinpoint the code where the problems are introduced. The framework does not aim to automatically exploit or reduce load imbalance unlike prior approaches. Instead, LIME provides highly accurate information to programmers about what causes the imbalance and where in the code it is introduced, with the goal to minimize trial-and-error diagnosis and the programming effort needed to alleviate the performance problem in parallel applications.

1.2.2 Barrier Latency and TL-based Hardware Barriers

Second scalability limiter I am addressing is the barrier. Barrier synchronization is useful building block for parallel programming that makes it possible to implement "meeting-point" between multiple or all threads. However, with growing numbers of cores within a chip and increasing wire latencies, barrier synchronization becomes increasingly challenging to achieve scalable performance.

The barrier problem is closely related with the load imbalance, the first scalability limiter presented in previous section. Ironically, as one gets more successful to reduce the load imbalance using the proposed LIME framework, the time to complete a barrier, i.e., barrier latency in the application becomes worse with non-scalable barriers. This is because the load imbalance reduction makes threads that used to arrive the barrier at different times (and hence caused load imbalance) arrive at similar times without much stall. That balanced arrival increases contention at the non-scalable barrier, resulting longer barrier latency.

Simplistic software-only implementations that maintain global arrival counter is the example of the non-scalable barrier. They suffer from serialization in accessing the global counter, so the barrier latency increases in proportion to the number of cores. Hierarchical barriers can reduce this contention, but require multiple rounds of point-to-point synchronization between cores and an inefficient broadcast via a shared memory location.

Hardware barrier networks provide much lower barrier latency and scalability but require a dedicated chip-spanning network whose wire latency has an undesirable technology-scaling trend. Worse, future many-core chips are likely to execute several parallel applications, virtual machines, etc., each of which may need a separate barrier network to synchronize its threads. Because the number of barrier networks (and their total cost) is a design-time decision, designers must decide between a) providing multiple expensive barrier networks that may be unused and b) not providing enough of them and potentially forcing applications to fall back to a much slower software barrier implementation. To address the latency problem of software barrier and multiple-barrier support issues faced by traditional hardware barrier networks, I propose a new hardware barrier system that exploits the advantages of "transmission lines".

Transmission lines have been proposed as a low-latency solution for on-chip interconnects, e.g. for express links in the on-chip network [21, 53] or for connecting distant banks in large caches [9]. In addition to providing much lower signaling latencies than traditional wires, transmission lines can be used to transmit multiple modulated signals at different RF frequencies. These advantages are tempered by increased cost and significant signal processing delays for demodulation and decoding of RF signals.

In this thesis, I propose a hardware barrier network, called TLSync, that allocates a relatively small part of the transmission line's available spectrum to each barrier group, allowing a single chip-spanning transmission-line network to support many (tens) of barriers simultaneously. In this barrier implementation, signal processing latencies are minimized by not relying on data transmission. Instead, TLSync barrier uses a simple signal presence test to determine when all cores in the group have arrived to the barrier. In essence, a TLSync barrier performs a wired-AND operation using the presence/absence of a "tone" at the barrier's allocated frequency to replace the traditional signaling using high and low voltage level on a wire. Each core transmits into the transmission line a low-power "tone" at its barrier's allocated frequency and stops transmitting when it finally arrives to the barrier. The receiver in each core is a simple tone detector tuned to the barrier's frequency, and a core can exit the barrier when this receiver no longer detects a "tone" at the barrier's frequency.

I provide an analysis of how the underlying chip-spanning transmission-line network can be designed to obtain accurate latency estimates for the proposed mechanism. This analysis offers several important insights on the tradeoffs that I face in using transmission lines (TLs) for on-chip transmission of RF signals. The key finding is that the end-to-end latency of RF transmission is dominated by fundamental signal processing delays that must be accounted for, and that careful modeling of TLs is necessary to account for severe signal weakening and degradation when multiple transmitters and/or receivers are connected along one transmission line.

1.2.3 Increasing On-chip Latency and Hybrid TL Interconnect

The last scalability limiter I explore in this thesis is the increasing packet latency in on-chip network. With the steadily growing number of on-chip cores, a shared bus has reached the point where it no longer provides sufficient bandwidth for the coherence traffic. As a result, many-core chip recently adopts a packet-switched network-on-chip (NoCs), e.g. a mesh or torus [26] that provides better throughput.

In such NoCs, each link is short and its length scales down with technology, so latency and energy consumption scale well if most traffic is local. Further, the aggregate bandwidth of all the links scales with the number of cores. However, switched networks come at a significant latency cost for traffic between far-apart cores—these messages traverse more switches when the core count grows, in addition to longer wire delays due to technology scaling. This increasing on-chip latency can be a serious obstacle to achieve good scalability performance for parallel applications because in usual, scalable applications try to fully utilize all cores in a chip, incurring communications between far-apart cores that cause significant latency disadvantage.

One viable solution for the worsening NoC latency is to use transmission lines in packet transmission. As denoted in TLSync, the signals propagation is close to the speed of light. However, TLs consume far more metal area than traditional wires, so the same-area TL-based NoC typically either provides far less throughput or requires very sophisticated signaling to improve this throughput.

I overcome this physical limitation of TL using an observation that the latency advantage of TLs over wires is the greatest for long-distance packets. Local traffic (which often represents the majority of the traffic) sees a relatively low latency even in a traditional mesh based NoC and it has little benefit from using costly TLs. Guided by this insight, a low-cost unidirectional TL (UTL) ring interconnect is designed. The

UTL ring provides very low maximum propagation latency (e.g. 2 ns for a 64-core chip), uses simple signaling and has a very efficient and simple arbitration mechanism. However, this ring has limited throughput, so I use it *together* with a traditional switched NoC, by judiciously steering each packet to the ring or the packet-switched interconnect. Evaluation shows that the TL hybrid interconnect can help parallel applications to run faster with more cores, improving execution time by 12.4% on average compared to the case where the mesh interconnect is used alone.

1.3 Thesis Statement

With the accelerating technology improvements, the number of available cores in a processor steadily increases. As a result, it is inevitable for application developers to exploit parallelism for better application performance. However, writing a fast parallel application is challenging due to many obstacles called *scalability limiters*.

This thesis addresses the scalability limiters to assist development of well-performing parallel applications. The scalability limiters occur from a variety of reasons, such as a defect in application design and incapability of the underlying hardware that executes the application. To effectively address those limiters, custom-built approach that deals with both software and hardware is necessary. Thus, I propose both a software framework and hardware improvements that are specifically devised to address three important scalability limiters: load imbalance, barrier latency and increasing on-chip packet latency.

This thesis proposes an initial exploration of collective approaches that are specifically targeted to address performance problems of parallel applications on many-core processors. The thesis proposes the following statement: ***Software frameworks and hardware techniques that address scalability limiters can cooperatively improve the performance of parallel applications on many-core processors.***

1.4 Thesis Overview

Chapter 3 explains the first software framework, *LIME*, that addresses load imbalance. The LIME framework is extended to incorporate applications running on heterogeneous cores in Chapter 4. Then, in Chapter 5, a new TL-based hardware barrier *TLSync* is proposed as a solution to current barriers that limits scalability. Further exploiting a transmission line design, I propose a new hybrid interconnect that explores the third scalability limiter, i.e., increased latency of switched on-chip networks in Chapter 6. I conclude in Chapter 7.

CHAPTER II

BACKGROUND AND RELATED WORK

2.1 Load Imbalance Debugging

Load imbalance has been one of the most serious problems in parallel and distributed systems. Load imbalance is described as the amount of wasted resource due to a lagging thread or machine, and many imbalance metrics have been proposed to characterize the load imbalance. One common metric is to define the imbalance as the difference between resource utilization of each entities using standard deviation between measured utilizations [5, 100]. Since this standard deviation depends on the measurement unit, often, coefficient of variation which is the ratio of the standard deviation over the mean value is used. This metric tells that if it is well load balanced, the dispersion of the measured utilization will be small, resulting smaller coefficient of variation. In [41], the load balance of parallel region is defined as the ratio between average and maximum efficiency across all processors. Likely to the first metric, this ratio denotes how the load is well distributed between different processors.

Performance debugging has long been studied in distributed systems. Much work focuses on finding the causal trace, or the trace with the longest path through a distributed system; this is analogous to the slowest thread in this study. The causal trace naturally tells programmers where to focus optimization efforts. LIME shares the same goal as this work, but works in a different domain. One notable study is the performance debugging method proposed by Aguilera et al. [3]—they use a black box approach that finds the causal trace without any knowledge of the system, and a signal processing technique called convolution to infer causal relationships. LIME collects profile data without a programmer's intervention, and uses clustering and regression analysis to infer causal relations between events and load imbalance.

A number of tools exist to detect and measure parallel overheads and inefficiency (i.e., idleness). One recent example is from Tallent et al. [92]. Their goal is to pinpoint where parallel bottlenecks occur, and classify bottlenecks as overhead or idleness. Gamblin et al. [34] proposes load balance measurement framework for large-scale systems. To avoid excessive overhead and perturbation associated with the system-wide measurement, they exploit wavelet analysis to compact profiled data.

This work is largely orthogonal to those tools—once a programmer knows that a problem exists, they can use this framework to help find the cause for the bottlenecks they have detected. Tallent et al. also analyzed lock contention [93]. This study shares a lot in common with LIME framework since they try to detect the cause of lock contention and identify the lock holder to blame. LIME focuses on barriers rather than locks and provides more direct information about where in the code the problem arises.

Also, many studies have tried to optimize performance and energy in the presence of load imbalance. Thrifty Barrier [60] predicts how much slack (i.e., idle time) each thread will have using a history-based predictor and save power by putting non-critical threads into a sleep state. Meeting Points [18] uses a different predictor that counts thread deviation at checkpoints called meeting points. It delays non-critical threads using dynamic voltage and frequency scaling to save power. It also attempts to accelerate the critical thread by prioritizing it.

The Thread Criticality Predictor [12] similarly predicts thread criticality based on adjusted cache miss rates, and prioritizes threads based on the prediction.

The automatic detection and prioritization of critical threads saves programmers effort, yet it only reduces load imbalance by a limited amount. Instead, LIME help programmers find and permanently fix imbalance problems in applications regardless of how severe the problems are; this can (and usually does) have a dramatic performance impact.

2.2 Hardware Support for Barriers

Researchers have long acknowledged the importance of efficient barrier mechanisms in highly-parallel systems, and many past supercomputer or massively parallel multicomputer systems had dedicated fast barrier hardware. Examples include a combining omega-switch network for fast fetch-and-add (barrier counter increment) in the NYU Ultracomputer [37], a single-stage combining network [40], and a dedicated bus for test-and-set messages in Sequent systems [8], dedicated networks in CM-5 [59] and Cray T3D [24], router extensions to support a virtual barrier tree in Cray T3E [86], and "global interrupt network" wired-OR in Blue Gene/L [4]. In the chip-multiprocessor arena, the MIT Multi-ALU processor [49] has a single-cycle barrier instruction implemented using global wires (that still had single-cycle latencies at the time). These dedicated barrier networks can only support one barrier at a time, whereas future many-core systems will likely execute several (potentially many) multi-threaded applications, each with its own barrier synchronization. Virtual barrier networks, such as the one in T3E, possibly bolstered with additional express links [84], are the exception to this—they can provide multiple virtual barrier trees, albeit at a cost of increased contention and latency. Beckmann and Polychronopolous [10] proposed a dedicated global interconnect for barriers that uses a register with zero-detect logic. However, the extension of this design for multiple concurrent barriers replicates much of the cost for each supported barrier, an approach that does not scale to large number of cores that may need many (ten or more) concurrent barriers. Sampson et al. introduce Barrier Filter [80], a lightweight barrier mechanism that uses cache invalidation as a barrier arrival signal and starves the cache fill until all threads arrive to the barrier. Unfortunately, the invalidation traffic to the centralized filter (which determines when all cores have arrived) can be a problem in many-core processors.

For future many-core chips, all these barrier networks also face the problem of increasing latencies for global wires. A barrier network spans the entire chip, so the wire length traversed by arrival and notification signals is expected to slowly increase as the total chip size increases, while the wire latency per unit distance quickly gets worse [43].

2.2.1 Transmission Lines

With growing numbers of cores per chip and increasing wire latencies, barrier synchronization becomes increasingly challenging even with the dedicated hardware supports. This is because the signal in traditional wires changes the potential of the entire wire whose capacitance is proportional to its length. Furthermore, future technology scaling trends [43] are expected to significantly increase the delays per unit length which are already long.

Transmission lines (TLs) have been proposed as a low-latency alternative to traditional global wires. In contrast to the traditional wire, TLs propagate signals as electromagnetic waves, at speeds that are close to the speed of light in the conductor material. This allows signals to cross the entire chip with sub-nanosecond propagation latencies, at least an order of magnitude faster than with traditional wires. This makes TLs very attractive for on-chip communication. However, TLs have several constraints that limit their usability: 1) they are more expensive than traditional wires because they are much wider (often by an order of magnitude or more) and occupy several (two or three) metal layers, 2) design of TLs requires much more modeling and planning because high frequency signals are significantly affected by the TL's shape (e.g., bends and turns) and implementation (e.g., material and thickness of the metal and insulator), 3) TLs require more sophisticated circuitry to transmit and receive their signals—instead of directly connecting logic gate inputs to a wire, TL receivers range from sense amplifiers to complete radio-frequency (RF) demodulation and decoding, depending on the type of the signal that is being used, and 4) data rates that can be achieved with relatively simple receivers (e.g., sense amplifiers) are limited.

That said, TLs will unlikely replace the entire global wires. Instead, TLs are likely to be used strategically, e.g. to provide low-latency global connections for cache banks in very large caches [9], "express" long-distance network-on-chip (NoC) links [21, 53], or as a chip-wide frequency-multiplexed bus [20].

Recently, a global network of TLs with baseband (non-modulated) signals has been proposed for barrier synchronization [1]. This hardware barrier has a lower latency than one built out of traditional wires. However, this scheme can still support only one barrier group at a time. Considering the large cost of TLs, it is unlikely that future chips will include multiple such global networks in order to support multiple barriers.

2.3 On-chip Network: Topologies and Latencies

Ever since more than one core is packed in one chip, various types of networks have been proposed for on-chip interconnects. Generally, the networks can be classified into three categories: unswitched (typically bus-based), packet-switched, and hybrid interconnects that exploit both. Unswitched interconnects broadcast all traffic on a shared medium, allowing simpler design than switched interconnects. However, as the number of cores and connections increases, the resulting contention becomes a major performance issue [38, 62]. Traditional buses also suffer from unfavorable wire delay and power trend as technology scales [32, 44, 90, 98].

In contrast, a packet-switched network typically employs short links between adjacent cores, which is efficient and has low-latency for local traffic. However, packets traveling to far-apart destinations have to go through many such links and on-chip routers. These routers are typically pipelined, which improves throughput, but further increases per-hop latency. This latency is somewhat alleviated by techniques such as lookahead routing [33], where routing calculation is performed one hop ahead to shorten the router's pipeline, or aggressive speculation [68] which also reduces pipeline length by speculatively performing switch allocation early. However, even with these advanced router microarchitectures, each traversed node still adds several cycles to the latency of a packet.

Hop count can be reduced by more advanced network topology. In a concentrated mesh [7], several nodes share each router, so fewer routing hops are necessary than the ones in a traditional mesh. In a flattened butterfly [50], hop count is reduced by providing richer physical connectivity to non-adjacent nodes. However, even with these efforts, growing core counts still lead to increase hop counts and wire latency, causing latencies of tens of cycles between far-apart cores.

The per-hop routing delays can be eliminated either by using a separate link for each pair of cores, which is extremely expensive for a large number of cores, or by using an unswitched interconnect. Unfortunately, as noted earlier, traditional unswitched interconnects (buses) suffer from both contention and poor scaling trends.

2.3.1 On-chip Transmission Lines

The latency and energy problems of wires in unswitched interconnects can be addressed with transmission lines (Section 2.2.1). For the specific TL-based design I use (Section 6.2.1), electromagnetic simulations show 7.5 ps/mm propagation speed, compared to ITRS projections [44] of 140 ps/mm for optimally repeated wires in 22 nm, and 150 ps/mm in 10 nm technology. However, TLs do not address the throughput problem of unswitched interconnects—in fact, they make it worse because the metal area occupied by a single TL is equivalent to tens of wires. Thus, a TL-based interconnect tends to have less throughput than the same-cost wire-based one.

2.3.2 Hybrid Interconnect and Steering

In contrast to my approach, where both networks are fully capable of delivering any message from its source to its destination, most prior work with multiple networks uses them hierarchically. Bourdaus [17] proposes hierarchical rings connecting local meshes to reduce global hop count. Local buses with global meshes are another hierarchical interconnect [27, 94] that can improve latency and power efficiency through communication locality. However, these hybrid proposals still incur significant routing delays for cross-chip packets in many-core processors, and growing wire delays are still a problem. In contrast, my approach combines a (traditional or advanced) switched NoC with a low-cost UTL ring that provides extremely low latency for selected (e.g. cross-chip) packets.

Other approaches to improve latency of many-core interconnect include attempts to hierarchically split chip-spanning bus into multiple local buses connected with point-to-point links [56]. Filtering broadcasts between segmented global and local buses further improves bus throughput [95] and the use of hierarchical (local and global) rings shows performance close to the packet-switched interconnect with improved scalability [32]. However, these interconnects still show strain in high-traffic applications, so they may only delay the transition to more scalable switched interconnects. In contrast, I embrace the transition to more scalable NoCs, but add TL ring to provide extremely low latency when needed. Furthermore, advances in scalable switched interconnects in terms of latency [39, 54] and power efficiency [31] may still result in significant latency differences depending on hop count.

Ring interconnects have been widely used in many systems for its simplicity with trivial routers and manageable latency with short point-to-point wires. IBM Power5 [89] and IBM Cell Architecture [47] exploits ring structure to connect multiple on-chip computing units with ring. Similarly, my approach with Transmission

Line adopts ring structure to ease the implementation with simple architecture. However, unlike the traditional ring structure, TL ring allows seamless signal propagation with novel couplers and pass-gates. When a pass-gate is in connected states, no delay is incurred to relay the signal. Using this, the ring can form a fast chain that starts and ends at the sending node. The coupler works like a ring resonator in nanophotonic interconnects [97] and allows every node on the chain to observe and relay the transmitted signal at the same time.

Steering messages in heterogeneous networks was explored in [22]. The heterogeneous network implements multiple interconnects with different characteristics in latency, bandwidth and power-efficiency by adjusting wire and repeater spacing. Then, coherence traffic is optimized to steer messages into different networks according to their properties (criticality, size, etc.). In contrast to steering between NoCs that are similar in nature, my approach steers between NoCs with very different latency, throughput and latency/hop characteristics.

Transmission lines (TLs) have been used for fast communications, e.g. low-latency barriers [1, 73], long-distance "express" links [21, 53] and connections between distant banks in large caches [9]. Shared-medium broadcast TL interconnects in [20] use optimizations to reduce on-chip traffic. While these optimizations defer the scalability problem, growth in core counts still inevitably leads to saturation of broadcast-type interconnects. In contrast, I rely on a switched network to provide throughput scaling, and use TL ring only for packets that benefit the most from it.

Alternative emerging approaches for low-latency interconnection include nanophotonics [23, 51, 76, 97] and on-chip wireless communications [35]. Nanophotonic NoCs can address the power and latency problem of the traditional NoC, though they require further investigation in their suitability to replace existing metal-based NoC architecture [101]. The application of TL to augment the existing NoCs differ with their approach in that I exploit already mature technologies and can ease the burden of installation. Hybrid wireless NoC expedites long-range communication but requires routing scheme and communication protocol that consume power and area overhead. Also, it has to overcome reliability and integration challenges.

Finally, a recent proposal describes a NoC composed of TL buses, with various optimizations to increase throughput [19]. However, such all-TL approaches increase throughput by using more TLs, complex encoding/decoding to embed more bits in the signal, and/or signaling at extremely high frequencies. My work is based on the insight that TLs provide little benefit for a large portion of the traffic (e.g. local traffic). That leads us to a low-cost approach of minimizing the total TL length and using binary pulse signaling at frequencies for which the required circuitry has already been demonstrated in existing CMOS technology [16, 74]. I then use the UTL ring to provide low latency for packets that can benefit from it while relying on a traditional scalable NoC for throughput requirement.

CHAPTER III

A FRAMEWORK FOR DEBUGGING LOAD IMBALANCE

3.1 Overview of LIME

In this chapter, I present LIME, a framework for debugging ***L***oad ***I***mbalance for ***M***ultithreaded ***E***xecutions. This framework utilizes statistical analysis in conjunction with control flow graph analysis on profiled data to *automatically* determine the nature of load imbalance problems and pinpoint the code where the problems are introduced.

To help explain LIME framework and the scalability limiter it addresses, I use the code example in Figure 1. This is an actual parallel section from SPLASH-2's *radix* benchmark [99]. The SPLASH-2 benchmark suite was extensively optimized over a decade ago by experts in both parallel programming and multi-processor hardware, and has been used to evaluate parallel performance of multi-processor and multi-core machines ever since.

This parallel section begins and ends with barriers, where each thread waits for all others to arrive before proceeding further. Any load imbalance will result in threads arriving at the end barrier (line 598) at different times, forcing early-arriving threads to wait (i.e., be idle) until the longest-running thread arrives.

Within the parallel section, each thread uses its private *MyNum* and *offset* values to decide which part of the parallel computation it should perform. Depending of these values, the threads may have different execution times, either by executing different code (due to differences in control flow) or by taking different amounts of time to execute the same code (due to differences in how the executed code interacts with

```
534    BARRIER(...);
535    if (MyNum != (...)) {
540        while ((offset & 0x1) != 0) {  ...  }
549        while ((offset & 0x1) != 0) {  ...  }
557        for (i = 0; i < radix; i++) {  ...  }
560    } else {
562    }
566    while ((offset & 0x1) != 0) {  offset=...  }
575    for(i = 0; i < radix; i++) {  ...  }
578    while (offset != 0) {
579        if ((offset & 0x1) != 0) {
582            for (i = 0; i < radix; i++) {  ...  }
585        }
589    }
590    for (i = 1; i < radix; i++) {  ...  }
594    if ((MyNum == 0) || (stats)) {  ...  }
598    BARRIER(...);
```

Figure 1: Code for sample parallel section from *radix*. All non-control-flow statements are removed.

the hardware). In Figure 1, there are several examples of possible control flow differences: if-then-else blocks at lines 535, 579, and 594 may cause only a subset of threads to execute the if-path (and for line 560, other threads to execute the else-path); loops at lines 540, 549, 557, 575, 578, 582, and 590 could all execute a different number of iterations for different threads. Nesting of loops and conditionals (e.g., line 582) can compound these differences. Additionally, threads may have differences in interacting with the system, such as branch predictor performance (some threads might have more predictable branch decision patterns than others) or cache performance (e.g., threads that access data already in the cache may have more cache hits). These differences are too numerous to point out even in this small example code because every branch, jump, load, store, etc. instruction in the compiled code may be, at least in theory, a potential source of these performance differences.

It may seem that a trained programmer can inspect the source code to identify potential causes of imbalance and repair them. However, this is a very labor-intensive and error-prone endeavor because of the sheer number of potential causes, and because

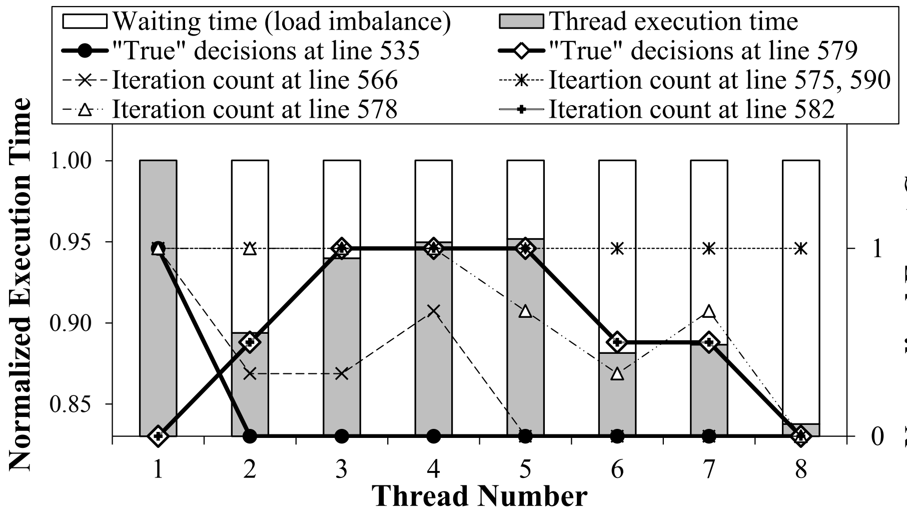


Figure 2: Execution time, imbalance, and thread behavior of key points in Figure 1. Points that cause load imbalance are shown with thicker lines.

of the complexity of understanding each cause and determining whether or not each is responsible for imbalance (and then repairing those that are).

An actual example of the threads' execution times for one dynamic instance of this parallel section is shown in Figure 2. The shaded part of each bar represents the useful execution time of each thread, normalized to the overall execution time of the parallel section. The white part of each bar represents the thread's waiting time at the end of the parallel section (line 598). Due to this waiting time, we have load imbalance of 0.051 according to our first definition in Section 2.1 (coefficient of variation, CoV) and 0.918 in second definition (ratio between average and maximum execution time). Note that the load imbalance shrinks as CoV decreases as well as the ratio increases.

Here, the discussion of possible causes of imbalance included two types of control flow causes—iteration counts of loops and decision counts of if-then-else blocks. These event counts are also shown in Figure 2, with each event's counts normalized to the maximum count for that event among all threads. From visual inspection of the graph, the decision at line 535 appears to cause imbalance between the first thread and the others. The differences in "true" decisions at line 579 appears to account for the remaining imbalance (note how well the useful execution time tracks this factor for threads 2 through 8). The loops at lines 575 and 590 have identical iteration counts in all threads, and thus cannot be causing any imbalance—code inspection reveals the same insight, because the value of *radix* is constant and the same for all threads. The loops at lines 566 and 578 do produce different iteration counts in different threads but this does not seem to correspond to actual imbalance. Finally, loop iteration counts at lines 540, 549, and 557 (not shown) have the same relationship with the imbalance that decision count at line 535 has, and loop iteration count at line 582 has the same behavior as the decision count for line 579.

LIME framework performs this kind of analysis automatically and quantitatively. For each thread in each parallel section, LIME measures execution time and various *event counts*. The event counts are dynamic decision counts for all static control flow decision points (control flow events), as well as dynamic counts for each static code location that causes machine-interaction events[1] (hardware events). Using this data, LIME's analysis framework determines how much imbalance exists, which control flow decisions and machine interaction events are related to the imbalance, and assigns scores that help programmers decide which cause of the imbalance to "attack" first.

The initial implementation of LIME includes two different profiling environments. The first implementation uses a cycle-accurate hardware simulator called SESC [79] that can be relatively easily extended to collect any desired machine-interaction event count; however, since it performs detailed simulation of a computer system, it is very slow and can only be used for parallel sections that execute quickly (e.g., with carefully designed small input sets). To overcome the speed limit, I implemented LIME with Pin [63], which is fast but can only accurately collect data for analysis of control

[1]Among machine-interaction events, I only experimented with cache misses because I expected them to be the only machine-interaction event that plays a significant role in creating load imbalance. As will be shown in Section 3.4.4, this turned out to not always be true. However, LIME's analysis treats all hardware events in the same way and I expect it to readily extend to other events (as long as they can be counted efficiently).

flow causes of imbalance. The simulator-based implementation was designed to let us experiment with collecting different events, and the Pin-based one to let us test LIME on larger input sets. For practical use, a purpose-built profiler could be employed to collect control flow events and key hardware performance counters more efficiently.

The analysis part of the framework processes profiling data from either profiling environment. It first clusters together events whose counts are highly correlated to each other. The purpose of this step is to group together events that seem to be related to the same potential cause of imbalance. For example, this step puts the decision count from line 535 and the iteration counts from lines 540, 549, and 557 in the same cluster because they are linearly related to each other (they have zero counts in all threads but one, so one of these event counts is equal to a constant times the execution count of another). Similarly, the iteration count from line 582 and the number of cache misses at lines 580–585 are in the same cluster as the "true" decision count from line 579 (this would not be true if threads had differing cache miss rates for lines 580–585).

Next, LIME finds the "leader" of each cluster. The purpose of this step is to identify the event that corresponds to "introducing" a potential cause of imbalance. In the two example clusters, the "true" decision counts from lines 535 and 579 are found to be the leaders of their respective clusters.

The next step in LIME uses multiple regression to find which cluster leaders are related to the imbalance in a statistically significant way, and to find the strength of that relationship. In this example, the "true" decision counts from lines 535 and 579 are the only cluster leaders to have a statistically significant relation to the load imbalance.

Finally, LIME ranks and reports the cluster leaders that are related to the imbalance. The report includes the score, the location in the code, and the corresponding cause of imbalance. In the example, LIME reports only two causes, both with relatively high scores: (1) threads take different paths at line 535, and (2) threads have different biases ("true" vs. "false" decision) for the if-then-else at line 579.

The following section provides a detailed description of the analysis framework. Subsequent sections describe the two profiling implementations, and an experimental evaluation of LIME's accuracy with examples that provide more intuition about how LIME works and how its results can be used by the programmer to reduce load imbalance.

3.2 LIME Analysis Framework

LIME's analysis starts with data gathered by one of the profiling implementations (see Section 3.3). The profiling data consists of 1) per-thread control flow event counts for each edge in the static control flow graph and 2) per-thread hardware event counts for each static instruction that can cause such an event.

3.2.1 Causality Analysis for Hardware Events

Before entering the main analysis routine, LIME conducts preprocessing on collected hardware events in order to establish causal relationships between related events. For example, an L1 cache miss can occur only when a memory access instruction is executed, an L2 cache miss can occur only when an L1 cache miss happens, etc. If the dynamic count for a particular hardware event differs among threads, this hierarchy among events allows us to split the "blame" for the difference between that event itself and the events that must precede it. For instance, when threads have different numbers of L1 misses at a particular static instruction, this may due to some threads executing that instruction more times (and thus having more L1 miss opportunities) or due to how the application interacts with the L1 cache. In the preprocessing step, LIME removes from subordinate hardware events (e.g. L1 misses) the contribution from their "superior" events (e.g. instruction's execution count) using the Gram-Schmidt process [36], leaving each event count only with the event's own contribution to variations among threads. This adjusted hardware event count is used instead of the naïve one throughout the LIME analysis.

3.2.2 Hierarchical Clustering

The second step in the LIME analysis is to cluster related events together. There are two commonly used clustering algorithms: hierarchical clustering and K-means clustering [45]. I use hierarchical clustering because it can automatically find an appropriate number of clusters for a given separation principle (clustering threshold) between clusters, whereas K-means yields a predetermined number of clusters. For the same reason, many projects on workload characterization [14, 48, 78] rely on hierarchical clustering to find benchmarks that have similar behavior.

Hierarchical clustering is performed in steps. Each step merges the two clusters that are "closest" according to a distance metric. Clustering ends when the distance between the two closest clusters is larger than a preset threshold.

In LIME, each event (control flow event or hardware event) is initially a cluster. I then compute a proximity matrix in which an element (i, j) represents the distance between 'cluster i' and 'cluster j'. At each step, I merge the two closest clusters and update the proximity matrix accordingly. The distance metric LIME uses is Pearson's correlation between the event's per-thread counts, because it effectively captures similarity in how the event count behaves in different threads. For linkage criteria, I used average linkage (UPGMA [70]), but other methods (single/complete-linkage) produced similar results.

LIME stops clustering when the largest correlation is <0.9. This threshold value provides the best results in most parallel sections I tested. The exceptions are *fmm* and *fluidanimate*, where I used threshold values of 0.8 and 0.6, respectively. A poorly chosen threshold affects the usefulness of the report—too-high of a threshold prevents merging of correlated event clusters, while too-low of a threshold results in clusters that contain unrelated events.

Clustering provides several benefits for further analysis:

- It results in a major reduction in the number of subjects for further analysis.
- It gathers highly co-linear events into one cluster, which improves accuracy of regression[2] in Section 3.2.5.
- It helps identify significant decision points in the program structure (e.g., branches where control flow differs between threads), as I explain in Section 3.2.4.

3.2.3 Classification of Clusters

Clusters are classified into two types: those that contain control flow events (control-flow clusters), and those with only hardware events (hardware event clusters).

For hardware event clusters, the absence of highly correlated control flow events indicates that different threads suffer the hardware events differently for the same code. Therefore, if any load imbalance is eventually attributed to the cluster, all the hardware events in the cluster are reported as contributing to that portion of the imbalance.

3.2.4 Finding Cluster Leaders

Within a control-flow cluster, control flow events are typically interdependent. To improve the usefulness of reported results, for each cluster, LIME discovers a *leader node*—a control flow instruction that steers program execution into the cluster. Intuitively, if the cluster is related to the imbalance, the leader node represents the code point where this imbalance is introduced.

[2]Statistical regression works poorly with collinear vectors.

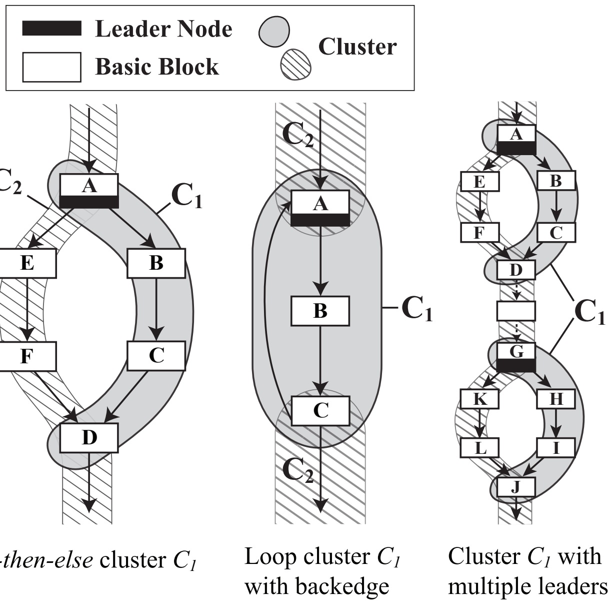

Leader Node
Basic Block
Cluster
C2
C1
A
B
C
D
E
F
G
H
I
J
K
L
if-then-else cluster C1
Loop cluster C1 with backedge
Cluster C1 with multiple leaders

Figure 3: Example of clusters and leader nodes

Leader nodes are important because they are the decision points that change thread execution characteristics. They are the points in the program of most interest to the programmer: by inspecting the code that affects the leader node's decision, the programmer can typically find the high-level reason for the imbalance.

To formally define a leader node, assume a control flow graph of a program has vertices V and edges E. The leader node of a cluster C is a vertex v in the control flow graph such that all incoming edges to v *except backedges* have source vertices outside the cluster, i.e., $v \in C$ such that $\forall(s, v) \in E$, $s \notin C$.

Examples of typical clusters and their leader nodes are shown in Figure 3. The leftmost example shows a cluster whose leader is an *if*-statement. The middle example shows why a backedge restriction is needed in the leader definition—it allows a loop cluster to have a leader (the loop entry point). The rightmost example shows a cluster with two leaders (A and G)—this typically occurs when the same control flow decision is made in more than one code point.

After finding leader nodes for each cluster, each leader node is assigned a score according to its significance in creating load imbalance. First, LIME computes, for each edge, the Pearson's correlation coefficient between that edge and the execution time. The score of a leader node is the *difference* in this correlation between the node's incoming and outgoing edges (again, ignoring backedges). For example, if n threads have execution times $T = (t_1, t_2, \cdots, t_n)$ and a leader node v has incoming edge counts $ie_{v_1}, ie_{v_2}, \cdots, ie_{v_i}$, and outgoing edge counts $oe_{v_1}, oe_{v_2}, \cdots, oe_{v_j}$, the score s_v of the leader node v is

$$s_v = \max_x\bigl(corr\left(oe_{v_x}, T\right)\bigr) - \max_y\bigl(corr\left(ie_{v_y}, T\right)\bigr) \tag{1}$$

where $corr(a, b)$ is the Pearson's correlation coefficient between vectors a and b.

Intuitively, score s_v measures the amount of correlation the leader node incurs in regard to the overall imbalance. A high score means that the node converts events that are unrelated to load imbalance into events that are highly correlated to the imbalance.

3.2.5 Multiple Regression

Multiple regression analysis [30] is a statistical technique that estimates the linear relationships between a dependent variable and one or more independent variables. In LIME, the dependent variable is the vector of per-thread execution times, and the independent variables are the clusters. If a cluster appears to be a good predictor of execution time, one can infer with high confidence that the cluster is responsible for the differences in execution time between threads.

For the regression analysis, it is needed to combine event counts of all events in the cluster so that the cluster behaves like a single event. For this, LIME uses averaged Z-scores [58] of events in a cluster. A Z-score standardizes all events to have the same average and same variation, so that all events carry equal weight in determining the cluster's overall "event count", regardless of the actual absolute event counts for each event. For each event with a mean per-thread dynamic count μ and a standard deviation σ, for each per-thread dynamic count x, the Z-score is $z = \frac{x-\mu}{\sigma}$. After this step, regression proceeds by treating each cluster as a single "event" whose per-thread "event counts" are the cluster's per-thread Z-scores.

During regression analysis, LIME excludes clusters that are statistically redundant or insignificant in building a regression model to explain the execution time. I use the forward selection method [30] to choose which clusters to include in the model. The method selects clusters based on their unique contribution to the variance in the dependent variable (execution time). LIME iteratively adds the selected clusters to its regression model until none of the remaining clusters is statistically significant (determined by F-test [61]).

Multiple regression analysis computes a standardized coefficient, β_C, for each cluster C, which represents how sensitive execution time is to that cluster. That is, the regression computes the values of β_{C_i} to best fit $T \approx \Sigma\beta_{C_i} \cdot C_i$, where T is the vector of per-thread execution times and C_i is the vector of averaged Z-scores for cluster i across all events in the cluster. LIME uses the β_C values as a measure of a cluster's importance for load imbalance as follows.

For each control-flow leader node and hardware event, LIME computes a *final score*—an estimate of how responsible that node/event is for load imbalance. The score is based on regression results and, for control-flow clusters, on leader node importance scores. For a leader node v_i in control-flow C_j, the final score fs_{v_i} is

$$fs_{v_i} = \beta_{C_j} \cdot s_{v_i} \qquad (2)$$

where s_{v_i} is the score (from Equation 1) for leader node v_i, and β_{C_j} is the standardized coefficient (from regression) for cluster C_j. For a hardware event (e.g., cache miss) h_i in a hardware-event cluster C_j, the final score fs_{h_i} is equal to β_{C_j}.

Figure 4 shows an example of computing final scores for two control-flow clusters. Cluster C_1 has one leader node A, while cluster C_2 has two leader nodes A and G. Note that node A is a leader in both clusters.

In this example, node G from cluster C_2 has the highest final score (i.e., a leader

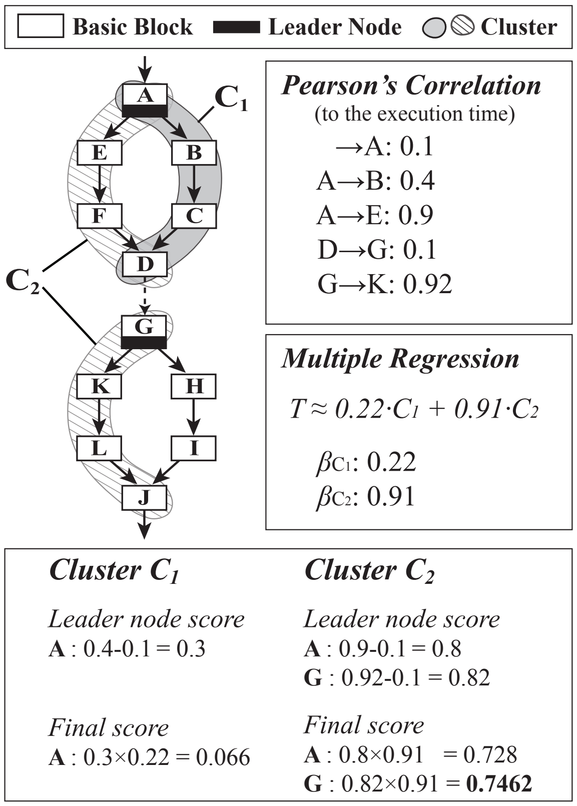


Figure 4: Example of final score computation

node score of 0.82 and a β_{C_2} of 0.91 combine to give a score of 0.7462). Therefore, one can conclude that cluster C_2 is important in explaining the imbalance, and that node G is the prime suspect for introducing the imbalance.

3.2.6 Reporting to the Programmer

In general, a program runs a static parallel section multiple times during its dynamic execution; LIME analysis operates on each dynamic instance independently because imbalance and other characteristics may vary across instances. Since programmers prefer feedback on static code, LIME then combines results from all dynamic instances of each parallel section using a weighted average, with load imbalance of a dynamic instance serving as its weight. For each leader node in a control-flow cluster, and for each hardware event (e.g., cache misses at static code location Y) in a hardware-only cluster, LIME presents to the programmer its static code location, type of event, and weighted score.

3.3 Implementation

The LIME framework consists of two parts: (1) collection of profiling data, and (2) analysis of the data. My contribution is primarily in the LIME analyzer, and my goal is to demonstrate the utility of the novel analysis techniques. I built the analyzer in C++ using the armadillo linear algebra library [81].

There is a large body of prior work on profilers. LIME framework can make use of any data profiler that can collect the required control flow and hardware event counts. For the prototype tool, I implemented two different versions of the profiler.

The first profiler is built on Pin [63], a binary instrumentation tool. With this profiler, one can collect control flow events, but no hardware events. I instrument synchronization points such as barriers to identify parallel sections in programs. I also instrument all branch instructions, and gather edge (control flow event) counts using a hashmap structure. When branch instruction b is executed and the previous branch was p, I increment the entry for edge (p, b). Since dynamic instrumentation severely distorts execution time, I use instruction count as an approximation to the execution time. When a dynamic parallel section ends (i.e., a thread enters a barrier), recorded data is exported to output files for later processing.

The second profiler is built on SESC [79], a cycle-accurate computer system simulator. This simulator functionally emulates an application and uses the instruction stream to drive a detailed model of the caches, memory, and processor cores, including the key microarchitectural structures (e.g., the instruction queue and reorder buffer, which allow for out-of-order execution). This simulator is routinely used by computer architects to evaluate architectural and microarchitectural proposals. I use the simulator to collect cache miss event counts and control flow edge counts simultaneously. Event recording and data exporting is very similar to the Pin-based tool (e.g., control

Table 1: Description of applications and parallel sections used in the experiments. Parallel section denotes the location of the *pthread_barrier_wait* or *pthread_join* call that delimits the parallel section.

Benchmark	Suite	Parallel Section	Type	Input Size
barnes	SPLASH-2	code.C:735	Barrier	16,384 particles
radiosity	SPLASH-2	rad_main.C:861	Barrier	room
volrend	SPLASH-2	main.C:291	Barrier	head-scaleddown4
fmm	SPLASH-2	fmm.C:298	Barrier	16,384 particles
water-spatial	SPLASH-2	interf.C:201	Barrier	3,375 molecules, 3 steps
ocean-cp	SPLASH-2	slave2.C:812	Barrier	514×514 grid
radix	SPLASH-2	radix.C:512	Barrier	4,194,304 integers
lu-cb	SPLASH-2	lu.C:618	Barrier	512×512 matrix, 16×16 block
fft	SPLASH-2	fft.C:646	Barrier	1,048,576 data points
blackscholes	PARSEC	blackscholes.c:444	Thread Join	4,096 options
fluidanimate	PARSEC	pthreads.cpp:1135	Barrier	36,504 particles
swaptions	PARSEC	HJM_Securities.cpp:299	Thread Join	64 swaptions, 20,000 simulations
canneal	PARSEC	annealer_thread.cpp:90	Barrier	10,000 swaps/step, 32 steps
streamcluster	PARSEC	streamcluster.cpp:706	Barrier	4,096 points

flow edges are counted with a hashmap). The primary difference is that SESC-based tool can accurately measure execution cycles and hardware event counts, but is much slower than the Pin-based version.

3.4 *Experiments and Results*

In this section, I present experimental setup and the output of LIME analysis framework, and then verify the output to show that LIME reports imbalance-related performance bugs accurately.

3.4.1 Experimental Setup

LIME is tested using data gathered from 14 parallel sections in multithreaded applications from SPLASH-2 [99] and PARSEC [15] benchmark suites. For runs with the SESC-based profiler, a typical multi-core general purpose processor is simulated with cores at 1GHz, 32KB each of data and instruction cache per core, and 64MB of shared cache. Since the accuracy of LIME depends on the number of threads (i.e., sample points), I run LIME analysis on a varying number of cores (8, 16, 32 and 64) to verify that the scheme can find performance bugs accurately across a wide range of available cores. In all configurations, one application thread is pinned on each core.

Table 2: Load imbalance occurred in benchmark applications for varying number of threads. Coefficient of Variation and Average/Max Ratio are weighted averages of per instance values.

Benchmark	**Imbalance**				**Coefficient of Variation**				**Average/Max Ratio**			
	8 Core	16 Core	32 Core	64 Core	8 Core	16 Core	32 Core	64 Core	8 Core	16 Core	32 Core	64 Core
lu-cb	28.4%	53.1%	92.7%	88.5%	0.16	0.35	0.81	0.84	0.86	0.75	0.64	0.61
water-spatial	61.7%	69.2%	73.8%	76.3%	0.43	0.48	0.54	0.58	0.53	0.46	0.40	0.35
fmm	13.1%	16.7%	26.4%	34.5%	0.06	0.07	0.11	0.15	0.92	0.89	0.82	0.72
volrend	15.2%	15.1%	23.3%	29.8%	2.64	3.85	5.49	7.33	0.13	0.06	0.03	0.02
canneal	4.1%	8.6%	14.9%	23.8%	0.01	0.02	0.04	0.08	0.98	0.96	0.93	0.89
fluidanimate	9.9%	15.0%	18.6%	22.7%	0.12	0.17	0.22	0.28	0.85	0.78	0.70	0.55
barnes	9.9%	11.3%	13.4%	15.7%	0.04	0.05	0.05	0.06	0.96	0.95	0.93	0.93
blackscholes	0.2%	0.8%	4.5%	10.6%	0.00	0.00	0.01	0.03	1.00	1.00	0.97	0.95
swaptions	1.1%	2.1%	2.1%	4.9%	0.00	0.01	0.01	0.01	1.00	1.00	1.00	0.98
fft	0.1%	0.1%	1.0%	1.6%	0.00	0.00	0.01	0.02	1.00	1.00	1.00	0.98
radiosity	0.3%	0.5%	0.8%	1.1%	2.32	2.69	2.30	1.36	0.14	0.09	0.07	0.09
streamcluster	0.3%	0.6%	1.0%	1.0%	2.62	3.82	5.42	7.42	0.13	0.06	0.03	0.02
radix	0.0%	0.0%	0.0%	0.2%	0.00	0.00	0.00	0.02	1.00	1.00	1.00	0.98
ocean-cp	0.0%	0.0%	0.0%	0.1%	0.01	0.00	0.00	0.01	0.99	1.00	0.99	0.98
Average	10.3%	13.8%	19.5%	22.2%	0.60	0.82	1.07	1.30	0.75	0.71	0.68	0.65

Table 1 summarizes the location and type of each parallel section.

In Table 2, imbalance of each parallel section is listed in decreasing order of imbalance on 64 cores. Coefficient of variation and Average/Max ratio are computed according to the description in Section 2.1. Imbalance is the average percentage of idle time threads spend in the parallel section in related to the total program execution time (i.e., $\frac{\sum max(T_i)-min(T_i)}{TotalExecutionTime}$, where T_i is the execution time of threads for instance i). The tested parallel sections cover a wide range of imbalance, from almost no imbalance (*radix* and *ocean-cp*) to over 90% imbalance (*lu-cb*).

3.4.2 Simulator-Based Profiling

Table 3 summarizes the results of LIME using the SESC-based profiler. The "Rpt" columns show the number of "important" events (those with a report score greater than 0.1) for both control flow events and cache miss events. The "Score" column shows the highest score reported among control flow and among cache miss events. The larger of the two highest scores is shown in bold font, to emphasize the event type that is more important according to LIME.

This result shows that LIME is able to draw a consistent and clear conclusion on

Table 3: Results of LIME analysis. The Rpt columns show the number of reported events with score greater than 0.1. The number of all reported events is listed in parentheses. The Score columns give the reported score for the highest-scored event of each type.

Benchmark	8 Cores				16 Cores				32 Cores				64 Cores			
	Ctrl flow		Cache miss		Ctrl flow		Cache miss		Ctrl flow		Cache miss		Ctrl flow		Cache miss	
	Rpt	Score	Rpt	Score	Rpt	Score	Rpt	Score	Rpt	Score	Rpt	Score	Rpt	Score	Rpt	Score
lu-cb	6 (6)	**0.99**	0 (0)	–	1 (6)	**1.00**	0 (0)	–	1 (5)	**1.00**	0 (0)	–	1 (1)	**1.00**	0 (0)	–
volrend	1 (1)	**1.00**	0 (0)	–	1 (1)	**1.00**	0 (0)	–	1 (1)	**1.00**	0 (0)	–	1 (1)	**1.00**	0 (0)	–
fmm	3 (5)	**0.64**	0 (8)	0.09	2 (3)	**0.38**	0 (10)	0.06	2 (3)	**0.41**	0 (4)	0.03	2 (7)	**0.21**	0 (0)	–
barnes	1 (1)	**0.97**	0 (3)	0.08	1 (5)	**1.06**	0 (2)	0.04	1 (6)	**0.93**	0 (0)	–	1 (3)	**0.92**	0 (4)	0.04
canneal	1 (4)	**1.23**	0 (10)	0.05	1 (4)	**0.83**	1 (7)	0.17	2 (3)	**0.77**	0 (9)	0.06	1 (3)	**0.57**	0 (25)	0.04
fluidanimate	1 (1)	**0.10**	0 (1)	0.01	1 (1)	**0.18**	0 (2)	0.03	1 (2)	**0.13**	0 (0)	–	1 (1)	**0.10**	0 (0)	–
water-spatial	2 (3)	**0.72**	0 (1)	0.03	1 (2)	**0.53**	1 (2)	0.11	2 (2)	**0.62**	0 (1)	0.07	2 (2)	**0.66**	0 (2)	0.04
streamcluster	2 (2)	**0.75**	0 (0)	–	2 (2)	**0.75**	0 (0)	–	2 (2)	**0.75**	0 (0)	–	2 (2)	**0.75**	0 (0)	–
radiosity	2 (2)	**1.00**	0 (0)	–	2 (2)	**1.00**	0 (0)	–	2 (2)	**0.99**	0 (2)	0.03	2 (4)	**0.89**	0 (3)	0.05
swaptions	6 (7)	**0.82**	1 (3)	0.22	6 (7)	**1.01**	0 (1)	0.05	6 (6)	**1.01**	0 (5)	0.06	6 (7)	**0.40**	4 (18)	0.28
blackscholes	1 (5)	0.57	4 (4)	**0.58**	1 (1)	**0.43**	4 (24)	0.35	1 (1)	0.32	4 (7)	**0.63**	0 (1)	0.03	4 (14)	**0.94**
fft	0 (0)	–	1 (2)	**1.14**	0 (1)	0.03	1 (3)	**0.73**	0 (0)	–	0 (0)	–	0 (1)	0.01	3 (4)	**0.69**
radix	0 (0)	–	0 (0)	–	0 (0)	–	0 (0)	–	0 (0)	–	0 (2)	0.04	0 (0)	–	1 (3)	**0.21**
ocean-cp	0 (0)	–	0 (0)	–	0 (2)	0.09	0 (0)	–	1 (1)	**0.31**	0 (2)	0.05	0 (1)	0.04	2 (4)	**0.39**

the important event type in most applications. For the top nine sections, control flow events are reported to cause the load imbalance, while for the next four, cache miss events are mainly reported for the cause of the imbalance.

LIME also consistently reports a small number of important events; this is important because it means the programmer should be able to inspect the spots in the source code that LIME reports. On average, for benchmarks with imbalance from control flow, it reports 1.9 important control flow instructions per benchmark. For benchmarks with imbalance from cache misses, on average LIME reports 2.7 important code points per benchmark.

3.4.3 Pin-Based Profiling

The limitations of the Pin-based implementation are that it introduces significant distortion in thread execution times and that it cannot accurately capture all the hardware interaction events that might be causing load imbalance. However, it can gather control flow edge information at speeds that allow "real-world" problem sizes.

To circumvent execution time distortion, in Pin-based profiling the instruction

Table 4: Results with Pin-based profiling.

Benchmark	Input Size	Slowdown	Ctrl Flow	
			Rpt	Score
LU	1K×1K matrix	21.1×	3 (10)	**0.97** (0.97)
	16K×16K matrix	23.9×	3 (6)	**0.93**
barnes	16,384 particles	84.1×	1 (6)	**0.96** (1.00)
	1,048,576 particles	88.7×	2 (5)	**0.97**
streamcluster	4,096 points	23.0×	2 (2)	**0.75** (0.50)
	1 million points	4.2×	2 (2)	**0.75**
radiosity	largeroom	62.6×	2 (3)	**1.00**
PLSA	100K sequence	127.0×	1 (13)	**1.36**

count is used as a proxy for execution time. This eliminates the possibility of identifying hardware-interaction events as causes of imbalance, but this profiler does not collect those events anyway. Further, the instruction count may not accurately represent the execution time. I validate that this profiler is useful in quickly identifying control flow sources of load imbalance. If hardware events trigger additional imbalance, a more expensive simulator-based approach can be used.

Overall, the control flow results of Pin-based profiling are very similar to those from the simulator. For brevity, Table 4 shows LIME results for only five parallel sections for "real" inputs (too large for simulation), including one from a benchmark (PLSA from bioParallel benchmark [46]) that is infeasible to run in simulation. Also shown are simulation-size inputs for three benchmarks for comparison, with scores from simulator-based profiling shown in parentheses. The profiling is done on an Intel Quad Core Xeon server using 8 threads (four cores, each with support for two threads).

3.4.4 Verifying Reported Cache Misses

While I have already shown that LIME framework consistently identifies a programmer-manageable number of causes of load imbalance, I now verify that the framework (with the SESC profiler) *correctly* identifies the causes of load imbalance, starting with cache misses. To verify that reported cache miss events are indeed causing load imbalance, one possible approach would be to try to reorganize the data structures

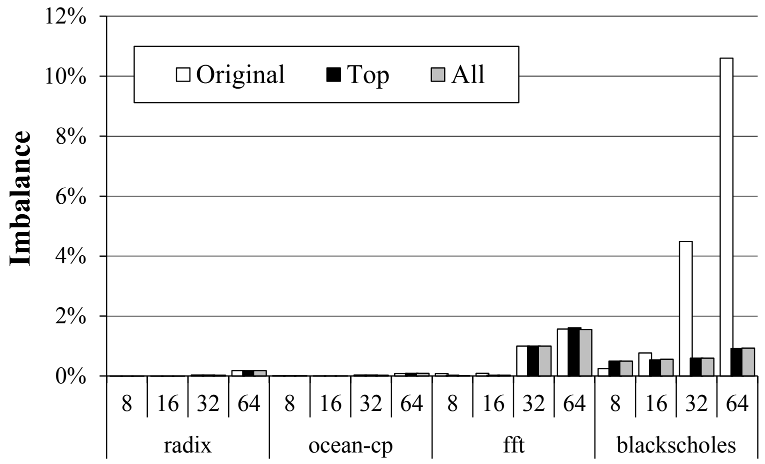


Figure 5: Imbalance reduction of each parallel section when reported cache misses are eliminated.

and algorithms in each application, re-evaluate parallel performance, and check if load imbalance has been reduced. However, I lack domain expertise and resources to make such extensive changes in so many applications. Further, the results would highly depend on how well one understood each application, data structure, and algorithm.

Instead, I use cycle-accurate simulation to artificially eliminate the reported cache misses, while leaving all other aspects of the execution intact. To "erase" misses from reported memory access instructions, I override the simulated cache behavior for that instruction to make each dynamic instance of that instruction into a cache hit. If the reported cache misses are indeed the source of the imbalance problem, this modified execution should have a dramatically reduced load imbalance.

Figure 5 shows the results of this simulation for applications in which LIME reports cache misses as the main cause of load imbalance. The black portion of each bar represents the resulting load imbalance when cache misses from the highest-scoring instructions are "erased" for each application. The shaded portion represents imbalance when "erasing" additional misses reported by LIME as statistically significant causes of load imbalance. The white portion represents imbalance when no cache misses are erased.

In Figure 5, *blackscholes* shows noticeable imbalance when thread count increases beyond 16 cores. For 32 and 64 cores, LIME reports 4 cache miss events with high confidence, resulting score of 0.63 and 0.94 for 32 and 64 cores, respectively. Load imbalance for *blackscholes* is reduced dramatically (86.7% and 91.2% for 32 and 64 cores, respectively) when LIME-reported cache misses are “erased”. It is important to note that, in all these simulations, I end up “erasing” misses from only 1.1% of all dynamic memory accesses, and the observed imbalance reduction is *not* caused simply by a dramatic reduction in execution time—in fact, the speedup in these runs mostly comes from reducing imbalance (making the slowest threads finish faster), with little performance benefit for the fastest threads.

For the rest three applications, reported load imbalance is not significant (<2%). LIME was not successful in finding a consistent cause of imbalance in such applications. For *radix* and *ocean-cp*, almost no imbalance (<0.2%) is detected for the configuration I used in this section. However, this may not be true in different configurations and inputs as explained in Section 3.4.6. For *fft*, neither control flow nor cache misses are the true cause. Further investigation reveals that the small imbalance is actually caused by cores having different success in getting access to the L1–L2 on-chip (back-side) bus when servicing cache misses—in these applications, bus arbitrator seems to be favoring some cores at the expense of others. However, unlike cache miss (and many other) events that can easily be attributed to particular instructions, such attribution for bus contention events is an open problem that is beyond the scope of this thesis.

3.4.5 Verifying Reported Control Flow Causes

To verify that LIME correctly reports control flow events that cause imbalance, I use a different methodology than for cache misses—control flow events cannot be “erased” without affecting many other aspects (including correctness) of program execution. Thus, in all parallel sections where LIME reported control flow code locations as a

Report from our analysis

No.	Address	Score	Code point (func.)
1	0x4018b4	0.880	lu.C:668 (lu)

Reported source lines in *lu.C*

```
668   if (BlockOwner(I, J) == MyNum) {   /* parcel out blocks */
669     B = a[K+J*nblocks];
670     C = a[I+J*nblocks];
671     bmod(A, B, C, strI, strJ, strK, strI, strK, strI);
672   }
```

```
556   long BlockOwner(long I, long J)
557   {
558     return((I + J) % P);    // P: number of threads
559   }
```

Figure 6: LIME report for *lu-cb*.

significant cause of imbalance (Table 3), I manually confirm that the location and nature of the problem was correct. In this thesis, I describe the result of analysis for three examples, selected to both illustrate different types of imbalance causes and to only require brief code fragments for explanation.

3.4.5.1 lu-cb

Among the parallel sections used, *lu-cb* has the most imbalance—the last-arriving thread takes over ten times longer than the first-arriving thread.

For the 32-core configuration, LIME framework reports only one static instruction, shown in Figure 6, as statistically significant cause of imbalance (at line 668). In the other three configurations, LIME also reports line 668 as the top-scoring (by a large margin) cause of imbalance.

The corresponding code point for the top-scoring instruction is also shown in Figure 6. This is an *if*-statement that parcels out blocks to threads using function *BlockOwner*, which returns the summed block coordinates modulo number of threads, P. Intuitively, this method of assigning blocks to threads should produce a balanced load. However, values of I and J vary in a range determined by the program's inputs—they may be small and/or not a multiple of P, causing uneven distribution of blocks

Report from our analysis

No.	Address	Score	Code point (func.)
1	0x4068ec	0.999	render.C:38 (Render)

Reported source lines in *render.C*

```
31    Render(int my_node)        /* assumes direction is +Z */
32    {
33       if (my_node == ROOT) {
34          Observer_Transform_Light_Vector();
35          Compute_Observer_Transformed_Highlight_Vector();
36       }
37       Ray_Trace(my_node);
38    }
```

main.C

```
298    Render(my_node);
300    if (my_node == ROOT) {
307         WriteGrayscaleTIFF(outfile, image_len[X], ... );
310         WriteGrayscaleTIFF(filename, image_len[X], ... );
312    }
```

Figure 7: LIME report for *volrend*.

to threads. For example, when I and J take values 1 through 15, the 225 blocks are distributed among 32 threads and ideally each thread should get about 7 blocks. The actual assignment using *(I+J)%P* turns out to assign only one block to threads 2 and 30, 2 blocks to threads 3 and 29, etc., giving 15 blocks to thread 16.

When line 558 in the *BlockOwner* function is changed to "return $(J\%Ncols) + (I\%Nrows) \times Ncols$;", where Ncols and Nrows are 8 and 4, respectively, for a 32-core configuration, the assignment of blocks to threads becomes more balanced and results in eliminating 61% of the original load imbalance. This confirms that imbalance was indeed introduced at the code point reported by LIME.

3.4.5.2 volrend

The second verification example is from *volrend*, which has up to 46.9% imbalance on 64 cores. The LIME report for 32 cores, summarized in Figure 7, suggests with high confidence that the return point of the function *Render* is the source of imbalance. At the first glance, this looks like a false report, but a closer inspection reveals that the

Report from our analysis

No.	Address	Score	Code point (func.)
1	0x405470	0.893	grav.C:116 -> grav.C:112
2	0x40548c	0.030	grav.C:113 (walksub)
3	0x4055cc	0.024	grav.C:136 -> grav.C:114

Reported source lines in *grav.C*

```
105   void walksub(nodeptr n, real dsq, long ProcessId)
106   {
107     nodeptr* nn;
              ⋮
112     if (subdivp(n, dsq, ProcessId)) { // First branch in walksub
113       if (Type(n) == CELL) {
114         for (nn = Subp(n); nn < Subp(n) + NSUB; nn++) {
115           if (*nn != NULL) {
116             walksub(*nn, dsq / 4.0, ProcessId);
117           }
118         }
```

Figure 8: LIME report for *barnes*.

problem is the mapping of the instruction to the line of source code because the compiler obfuscated the situation via inlining and other optimizations. When I examine the code in a disassembler, the reported control flow instruction is actually the *if*-statement at line 300 (immediately after the callsite for *Render*). This *if*-statement assigns extra work to the main thread. When compiled without optimizations, LIME reports the *if*-statement in line 300 correctly.

3.4.5.3 barnes

The third verification example is from *Barnes*, with 13.5% imbalance on the 32-core configuration. The LIME report for 32 cores, summarized in Figure 8, says the control flow edge from line 116 in *grav.C* to line 112 accounts for the imbalance. This edge corresponds to the recursive function call to *walksub*—*Barnes* implements the Barnes-Hut approach for the N-body problem, and *walksub* recursively traverses the primary data structure, a tree. Since LIME reports the tree traversal is imbalanced, this suggests that the tree itself is imbalanced. Further investigation shows that 86.5% of the total imbalance comes from the first dynamic instance of the parallel section,

Table 5: Description of small, medium and large inputs for benchmark applications.

Benchmark	Small Input	Medium Input	Large Input
barnes	16,384 particles	32,768 particles	262,144 particles
radiosity	room, 0.15 BF epsilon	room, 0.015 BF epsilon	room, 0.0015 BF epsilon
volrend	head-scaleddown4 , 20 steps	head-scaleddown2 , 50 steps	head-scaleddown2 , 100 steps
fmm	16,384 particles	65,536 particles	262,144 particles
water-spatial	3,375 molecules, 3 steps	8,000 molecules, 3 steps	32,768 molecules, 3 steps
ocean-cp	514×514 grid	1,026×1,026 grid	2,050×2,050 grid
radix	4,194,304 integers	16,777,216 integers	67,108,864 integers
lu-cb	512×512 matrix	1,024×1,024 matrix	2,048×2,048 matrix
fft	1,048,576 data points	4,194,304 data points	16,777,216 data points
blackscholes	4,096 options	16,384 options	65,536 options
fluidanimate	36,504 particles	102,850 particles	305,809 particles
swaptions	64 swaptions, 20,000 sim.	64 swaptions, 20,000 sim.	64 swaptions, 40,000 sim.
canneal	10,000 swaps/step, 32 steps	15,000 swaps/step, 64 steps	15,000 swaps/step, 128 steps
streamcluster	4,096 points	8,192 points	16,384 points

because the tree is skewed at the beginning of the run. The first Barnes-Hut iteration rebalances the tree and the load imbalance decreases.

3.4.6 Input Sensitivity of LIME

LIME relies on profiling to get information required for the statistical analysis to find the source of load imbalance. Since the profiling can only capture dynamic information that occurs in a specific execution, it is possible for LIME to fail to detect any source of load imbalance if the profiled data is insufficient or does not provide enough information to detect the load imbalance. For example, if an execution does not cover a specific path that induces imbalance, LIME cannot identify the path as the cause since it is not present in the profiled data. Moreover, imbalance of an application may increase or decrease as the input of the application or execution environment such as number of threads changes. For example, as denoted in Table 2, increasing thread count deteriorates imbalance severely (0.2% for 8 threads to 10.6% for 64 threads).

The limitations of dynamic profiling techniques and dynamic analysis impose limitation on LIME's ability to handle load imbalance problem. However, these limitations are open problems and have long been studied in software engineering community [11, 67, 82, 83]. It is beyond the scope of this thesis to address those problems,

Table 6: Load imbalance of benchmark applications for varying input size and thread counts.

Benchmark	8 Core		16 Core		32 Core			64 Core		
	Small	Medium	Small	Medium	Small	Medium	Large	Small	Medium	Large
lu-cb	28.3%	15.2%	53.2%	30.0%	92.6%	55.2%	31.1%	88.5%	94.4%	55.3%
water-spatial	61.6%	41.3%	69.2%	63.3%	73.7%	72.9%	47.8%	76.3%	77.0%	53.1%
fmm	13.0%	6.5%	16.8%	7.6%	26.4%	11.9%	7.3%	34.6%	17.3%	10.9%
volrend	15.2%	17.4%	15.1%	17.0%	23.3%	28.3%	29.5%	29.8%	41.7%	43.7%
canneal	4.1%	3.3%	8.6%	5.6%	14.9%	9.6%	8.6%	23.8%	14.1%	12.1%
fluidanimate	10.0%	5.7%	15.0%	6.5%	18.6%	9.0%	14.1%	22.6%	11.2%	17.6%
barnes	9.9%	8.2%	11.3%	9.8%	13.4%	12.0%	9.7%	15.7%	16.2%	14.5%
blackscholes	0.2%	0.0%	0.8%	0.4%	4.5%	2.8%	0.4%	10.6%	5.4%	2.6%
swaptions	1.1%	0.5%	2.1%	1.1%	2.1%	2.1%	2.1%	4.9%	4.9%	4.9%
fft	0.1%	0.2%	0.1%	1.1%	1.0%	1.6%	0.9%	1.6%	4.2%	3.0%
radiosity	0.3%	0.0%	0.5%	0.1%	0.8%	0.2%	0.0%	1.1%	0.4%	0.1%
streamcluster	0.3%	0.2%	0.6%	0.3%	1.0%	0.5%	0.2%	1.1%	0.7%	0.2%
radix	0.0%	0.3%	0.0%	0.1%	0.0%	0.4%	0.1%	0.2%	0.9%	0.6%
ocean-cp	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%	0.0%	0.1%
Average	10.3%	7.1%	13.8%	10.2%	19.5%	14.8%	10.9%	22.2%	20.6%	15.6%

and here I present the result of LIME on varying inputs to observe the sensitivity of LIME analysis as the profiled information changes.

Table 5 lists three different input parameters tested in this section. For PARSEC benchmark suite, predefined input sizes (sim-small, sim-medium and sim-large) are used. SPLASH-2 inputs are scaled up to increase application execution time.

In Table 6, the resulting load imbalance is listed. Note that the imbalance in general decreases as the input size increases because the total amount of job elements that threads execute grows with larger input. More job elements make it easier to distribute the elements across threads evenly.

LIME analysis generates consistent results for the modified input sets. Result of LIME analysis for 64 core configuration is listed in Table 7. For applications that control flow events cause load imbalance in Table 3, LIME analysis reported control flow events on program execution with larger input sets. Although scores vary, highly reported events identify the same code locations in all the applications except for *water-spatial*. It implies that LIME can correctly identify static sources of load imbalance even when the profiled data and the amount of imbalance change with different input sets.

Table 7: Results of LIME analysis for 64 core configuration on varying input sizes (Small, Medium and Large).

Benchmark	**Control Flow**						**Cache Miss**					
	Small		Medium		Large		Small		Medium		Large	
	Rpt	Score	Rpt	Score	Rpt	Score	Rpt	Score	Rpt	Score	Rpt	Score
lu-cb	1 (1)	**1.00**	1 (5)	**1.00**	1 (5)	**1.00**	0 (0)	–	0 (0)	–	0 (0)	–
volrend	1 (1)	**1.00**	1 (1)	**1.00**	1 (1)	**1.00**	0 (0)	–	0 (0)	–	0 (0)	–
fmm	2 (7)	**0.21**	2 (7)	**0.22**	6 (10)	**0.37**	0 (0)	–	0 (5)	0.02	0 (9)	0.08
barnes	1 (3)	**0.92**	1 (5)	**0.92**	1 (2)	**0.87**	0 (4)	0.04	0 (4)	0.03	0 (5)	0.02
canneal	1 (3)	**0.57**	1 (2)	**0.92**	1 (3)	**1.05**	0 (25)	0.04	0 (5)	0.04	0 (4)	0.02
fluidanimate	1 (1)	**0.10**	2 (5)	**0.44**	1 (2)	**0.14**	0 (0)	–	0 (2)	0.02	0 (0)	–
water-spatial	2 (2)	**0.66**	1 (2)	**0.11**	1 (1)	**1.00**	0 (2)	0.04	0 (1)	0.02	0 (4)	0.02
streamcluster	2 (2)	**0.75**	2 (2)	**0.75**	2 (2)	**0.66**	0 (0)	–	0 (0)	–	0 (0)	–
radiosity	2 (4)	**0.89**	2 (4)	**0.87**	2 (4)	**0.89**	0 (3)	0.05	0 (4)	0.06	1 (4)	0.11
swaptions	6 (7)	**0.40**	6 (7)	**0.40**	6 (8)	**0.38**	4 (18)	0.28	4 (18)	0.28	4 (19)	0.25
blackscholes	0 (1)	0.03	0 (1)	0.02	0 (0)	–	8 (14)	**0.94**	16 (22)	**0.89**	21 (24)	**0.96**
fft	0 (1)	0.01	0 (1)	0.03	0 (1)	0.08	3 (4)	**0.69**	2 (3)	**0.92**	4 (4)	**0.70**
radix	0 (0)	–	0 (0)	–	0 (0)	–	2 (3)	**0.21**	1 (3)	**1.00**	2 (2)	**1.00**
ocean-cp	0 (1)	0.04	1 (1)	0.14	0 (0)	–	2 (4)	**0.39**	1 (4)	**0.18**	1 (1)	**0.96**

LIME reports different code locations for *water-spatial*. The small input set reports outermost loop that computes intermolecular interaction as the source of imbalance, while the large input reports an inner loop. The medium input reports both inner and outer loops (with different scores). The discrepancy between the small and medium input set is because the small input, due to its small working set size, assign uneven number of job elements to operate on outer loops. The large input size assigns an equal amount of coarse-grained job elements that determine outer-loop iteration, allowing inner loop as the cause of load imbalance.

Applications with imbalance caused by cache misses exhibit varying amount of load imbalance with different input sizes. *blackscholes* shows rapidly decreasing imbalance while the input set grows. The load imbalance is caused by an excessive amount of conflict misses in private cache due to ill-designed data structures. With larger input set, the percentage of imbalance decreases as the computation time (which is balanced) grows faster than data loading time. Yet, LIME still correctly identifies the location of memory access location that induces load imbalance.

For *radix*, *fft* and *ocean-cp*, the reason of imbalance does not appear consistent among the various input sizes. Further investigation shows that the contention in shared cache caused sightly increased imbalance at *radix* and *ocean-cp* while that

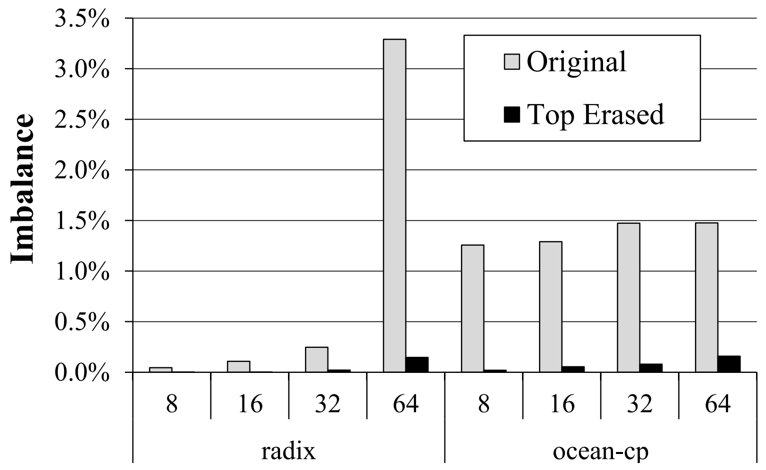


Figure 9: Imbalance of *radix* and *ocean-cp* applications on smaller cache configuration before and after we "erased" reported cache miss events from LIME. Top-erased means only top reported events are treated as "cache-hits"

in shared bus that connects private and shared cache caused imbalance at *fft*. To validate this observation, modified configuration with decreased cache sizes has been tested to exacerbate the contentions. Here, I halve private cache size (32KB to 16KB) and reduce shared cache size from 64MB to 4MB. As a result, applications show increased imbalance (depicted in Figure 9). LIME reports code locations that access data structure, and such locations exhibited excessive L2 cache (shared cache) misses, meaning that the contention in shared cache caused load imbalance for those applications. Fixing those cache misses by "erasing" them as explained in previous section dramatically reduces load imbalance (95% on average), which verifies that LIME can accurately pinpoint transient load imbalance caused by cache misses.

3.4.7 Scalability of LIME

LIME ran fast enough for this study. Yet, two types of concerns have been raised: (1) what happens to analysis time as the core count increases, and (2) what happens to analysis time as event count (i.e., program size) increases?

A single-threaded implementation of LIME analysis took an average of 1.4, 1.8, 3.2, and 5.0 seconds to analyze the parallel sections for 8, 16, 32, and 64 cores respectively on a 2.67GHz Intel Quad Core Xeon processor with 12GB memory. This time includes clustering, leader selection and regression. Since the analysis time grows in sub-linear proportion to the core count, I expect that a parallel implementation of LIME analysis will have a favorable scaling trend as the number of cores (for both application execution and analysis) increases.

The most time consuming part of LIME is the clustering part of the analysis—it accounts for over 76% of the average analysis time when no optimization is applied. A naive implementation of hierarchical clustering method has asymptotic complexity of $O(n^3)$ where n is the number of event counts. LIME merges $O(n)$ times to create a near-constant number of clusters. Here, all $O(n^2)$ entries in the new cluster proximity matrix is recomputed in each merge. This method can become a major problem when applying LIME to real applications. To accelerate the clustering algorithm, one can cache the proximity matrix and perform bulk clustering. When merging two clusters, only the proximity values involving those two clusters become useless. Therefore, I reuse (cache) the proximities and compute only $O(n)$ new values. The "bulk clustering" optimization merges multiple clusters per step. I track clusters that are very close to each other when compute the proximity matrix and merge all of them at once. In this study, I use a proximity threshold of 0.99 for bulk clustering. Bulk clustering reduces the number of merges, but runs the risk of producing less precise clustering. I did not experience any imprecise clustering in this experiment.

Figure 10 compares the speed of the clustering implementation with and without the acceleration technique using a log–log plot. While my optimized implementation is still $O(n^2)$ (bulk clustering does not reduce the asymptotic complexity), in practice it shows near-linear scaling with n. This demonstrates that LIME is scalable with a range of program sizes. I leave further optimization and fine-tuning for future work.

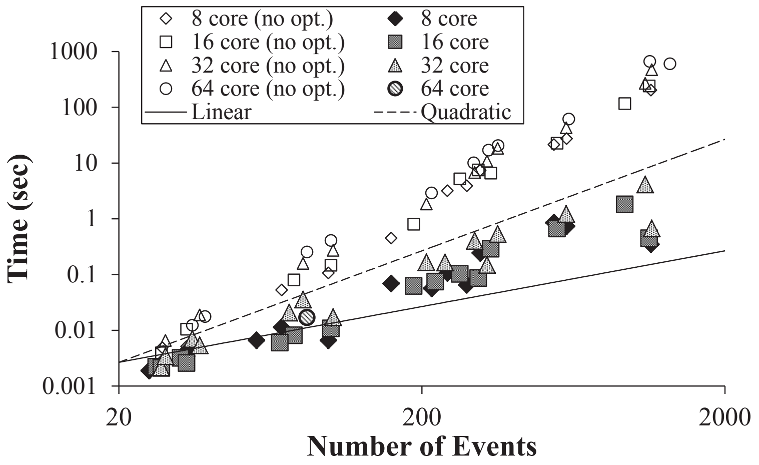


Figure 10: Clustering time vs. number of events. Each point represents clustering time of one parallel section. Note the logarithmic scale markings on each of the axes.

CHAPTER IV

LIME EXTENSION FOR HETEROGENEOUS CORES

In this chapter, I present an extension of LIME framework for heterogeneous systems.

4.1 Load Imbalance Debugging for Heterogeneous Cores

The LIME framework presented in the previous chapter exploits statistical techniques to identify the source of load imbalance. In particular, multiple regression and correlation coefficient are used to estimate how events contribute to the load imbalance. However, it was implicitly assumed that the cost of each event remains the same regardless of the core on which the event occurs. This assumption holds true if applications are run on a chip multiprocessor (CMP) that consists of homogeneous cores. However, with heterogeneous processors that have cores with different processing power or capabilities, the assumption is no longer valid and the LIME framework tends to report load imbalance that does not exist in reality and fail to correctly report the actual load imbalance.

Heterogeneous processors have long been an active research topic as a promising approach to low-power and efficient architectures. The integration of multiple heterogeneous cores in one chip makes it easier to meet various energy and performance goals than the use of homogeneous cores. For example, Heterogeneous Chip Multiprocessors [55] employ multiple generations of core designs to enhance single-thread performance while maintaining throughput advantage of CMP. Also, NVIDIA released Tegra 3 [71] processor that comes with a power efficient but lower-performance companion core called "battery-saver" along with main processing units for mobile computing. The "battery-saver" core handles low-power tasks that do not need much computing power, such as standby actions or music playback, to improve the overall power efficiency.

With these advantages, the demand for heterogeneous architectures is increasing. However, for those applications that are not optimized for heterogeneous processors, the load imbalance grows significantly (Figure 17). There are two reasons for the exacerbated load imbalance in heterogeneous environments. First, with heterogeneous-agnostic applications, the large and powerful cores complete much faster when they are given the same amount of work as the small and power-efficient cores (Figure 11b). These faster threads create load imbalance when blocked by a global synchronization such as a barrier. Second, the deviation of the program execution between the same types of cores incurs the imbalance within the big (or small) cores.

Many studies have shown that the load imbalance caused by the first reason can be mitigated by frameworks that perform heterogeneous-aware workload balancing such as Qilin [64]. However, even if we properly assign more work to the big cores, the imbalance still remains among the big (or small) cores because of the second reason. This remaining imbalance occurs *not* because the underlying cores are heterogeneous, *but* because the program itself incurs imbalanced execution between threads, and the frameworks for heterogeneous-aware load balancing cannot eliminate this imbalance. What is more, LIME framework presented in Chapter 3 is unable to operate with the data profiled from heterogeneous architectures.

Table 8 shows the result from LIME on homogeneous and heterogeneous cores. For applications that LIME used to successfully report performance bug points with homogeneous cores, it reports no outputs or very low scores with data profiled on heterogeneous cores. This is because the LIME is built on an implicit assumption—the underlying cores are all homogeneous, i.e., the performance and capability do not vary across cores—that breaks.

In this chapter, I propose an extension of LIME analysis framework that allows

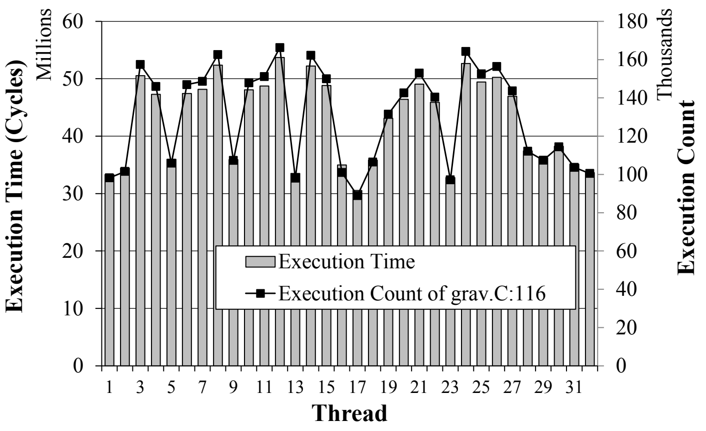
Table 8: Scores reported by LIME analysis for homogeneous cores (Section 3.4.2) and for heterogeneous cores. For heterogeneous cores, unmodified LIME failed to locate the cause of imbalance in many applications.

Benchmark	**Control Flow**				**Cache Miss**			
	Homogeneous		Heterogeneous		Homogeneous		Heterogeneous	
	Report	Score	Report	Score	Report	Score	Report	Score
lu-cb	1 (5)	**1.00**	1 (2)	**0.64**	0 (0)	–	1 (3)	0.11
volrend	1 (1)	**1.00**	1 (1)	**1.00**	0 (0)	–	0 (0)	–
fmm	2 (3)	**0.41**	0 (0)	-	0 (4)	0.03	35 (60)	**0.60**
barnes	1 (6)	**0.93**	0 (0)	-	0 (0)	–	21 (34)	**0.62**
canneal	2 (3)	**0.77**	0 (1)	0.01	0 (9)	0.06	3 (13)	**0.49**
fluidanimate	1 (2)	**0.13**	0 (0)	-	0 (0)	–	9 (16)	**0.56**
water-spatial	2 (2)	**0.62**	1 (1)	0.26	0 (1)	0.07	23 (42)	**0.60**
streamcluster	2 (2)	**0.75**	2 (2)	**0.75**	0 (0)	–	0 (0)	–
radiosity	2 (2)	**0.99**	2 (5)	**0.39**	0 (2)	0.03	0 (3)	0.07
swaptions	6 (6)	**1.01**	0 (2)	0.03	0 (5)	0.06	7 (7)	**0.26**
blackscholes	1 (1)	0.32	0 (1)	0.02	4 (7)	**0.63**	6 (13)	**0.83**
fft	0 (0)	–	0 (0)	-	0 (0)	–	13 (13)	**1.02**
radix	0 (0)	–	0 (0)	-	0 (2)	0.04	4 (4)	**1.00**
ocean-cp	1 (1)	**0.31**	0 (0)	-	0 (2)	0.05	0 (0)	–

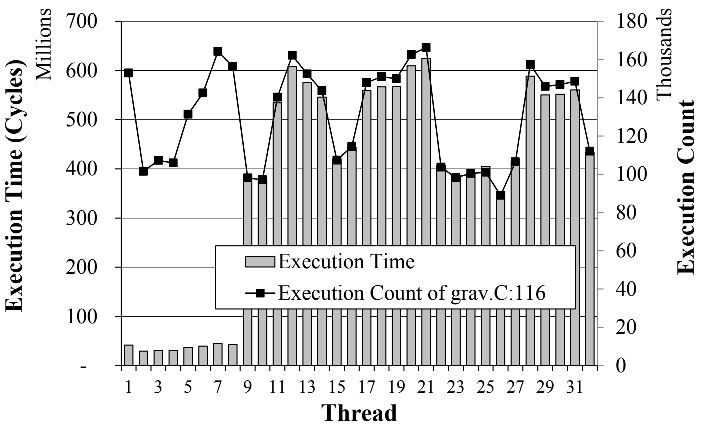
the debugging of the program-intrinsic imbalance caused by the second reason in heterogeneous architectures. The extension *adjusts the profiled data* from heterogeneous architectures to keep the homogeneous core assumption intact.

Figure 11 explains how the assumption breaks and why LIME fails in such heterogeneous environments. Figure 11a depicts execution time of threads when *barnes* from SPLASH-2 benchmark is run on CMP with homogeneous cores using configuration described in Section 3.4. Along with the execution time, execution count of line 116 from *grav.C* is shown in Figure 11. The line is a recursive function call that performs depth-first tree search and causes severe load imbalance for the application. As seen in the figure, the event count shows a good correlation with the execution time, denoting that the event plays an important role in the generation of the execution time imbalance between threads. Here, the "cost" of an event (*grav.C:116*) is roughly the same for all cores—if the event count of thread x_1 and x_2 differ by the same amount as thread y_1 and y_2, then the execution time difference is roughly the same regardless of x and y.

However, this "same cost" assumption does not hold true in Figure 11b where some of the cores (core 1–8) are capable to run faster (called "big" cores) than others



(a) Homogeneous Cores



(b) Heterogeneous Cores

Figure 11: Execution time and execution count of an event that causes load imbalance (*grav.C:116*) from *barnes* on homogeneous and heterogeneous cores.

(called "small" cores). In the heterogeneous configuration (Table 11), threads 1–8 that are run on core 1–8 exhibit much less execution time while the event counts of line 116 remain unchanged compared to the homogeneous case. As a result, the "cost" of event *grav.C:116* differs between threads on "big" cores and those on "small" cores. For example, thread 1 executes the line 50% more than thread 9 (155 thousands vs. 100 thousands) but runs 10 times faster, implying that the *grav.c:116* event cost on a small core is almost 15 times as expensive as on a big core.

Statistical analysis used in LIME did not accommodate the various event costs that arise in heterogeneous processors. LIME exploits multiple regression and correlation coefficient to identify a combination of events that linearly correlate well with

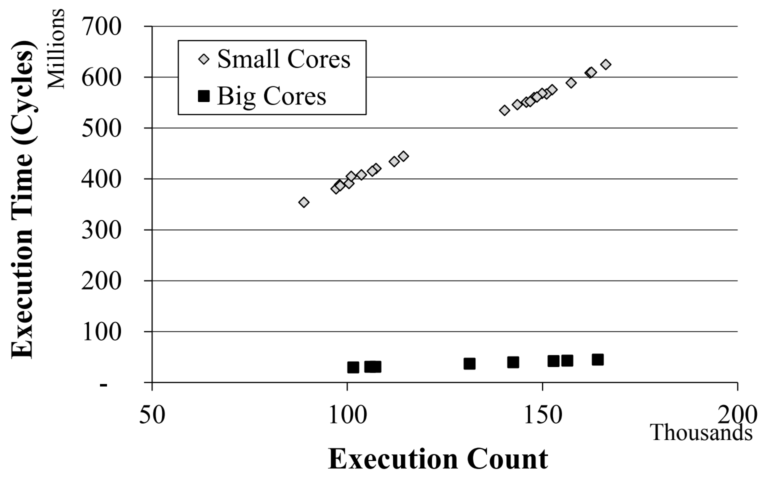


Figure 12: Execution time and event count of *grav.C:116* from heterogeneous cores. Note that it is not possible to find a common trend line for both cores.

the thread execution time. In essence, this analysis finds constant coefficients for a linear equation that describes the event count–execution time relationship, and those constant coefficients correspond to the cost of each event. However, with heterogeneous cores, the costs of events differ significantly between the big cores and the small cores, so they cannot be represented by one constant cost for each event. Figure 12 illustrates this by showing the trend line for small cores and for big cores. The slopes of these lines correspond to event cost, and we see that the lope for small cores is much steeper than that of the big cores. Here, the correlation between the event count of line 116 and execution time drops to 26.9%. The same line shows 99.9% correlation with the execution time with homogeneous cores.

In this chapter, I propose an extension of LIME analysis framework that allows multiple event costs from heterogeneous architectures. The extension *adjusts* profiled data from heterogeneous architecture to *compensate* for varying performance of different types of cores. For the adjustment, we can modify all the profiled event counts to equalize them, or change thread execution time. For simplicity, we choose to adjust thread execution time.

Figure 13 shows how the adjustment enables the application of LIME process

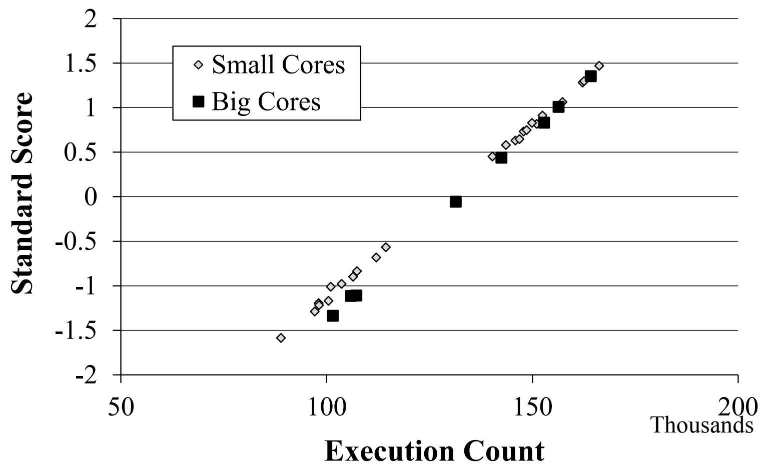


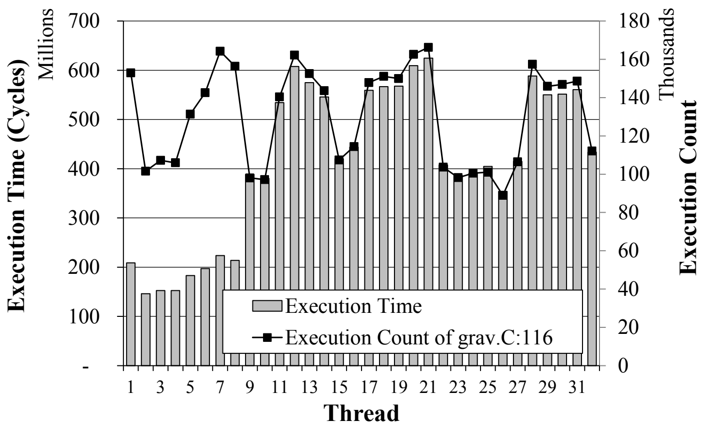
Figure 13: Adjusted execution time using standard score and event count of *grav.C:116* from heterogeneous cores.

on data from heterogeneous cores. Here, thread execution time is adjusted using standard score [52]. Standard score enables to compare the performance of different types of cores (big and small here) using the same mean and variation. For this *barnes* application, the score scales event costs from both big and small cores so that the correlation between execution time and imbalance causing event is restored. However, this is not always the case. In many applications, standard score fails to provide proper scaling. In Section 4.2, I explain why it fails and present better ways to adjust the thread execution time. Section 4.3 shows experiments and results of this extension.

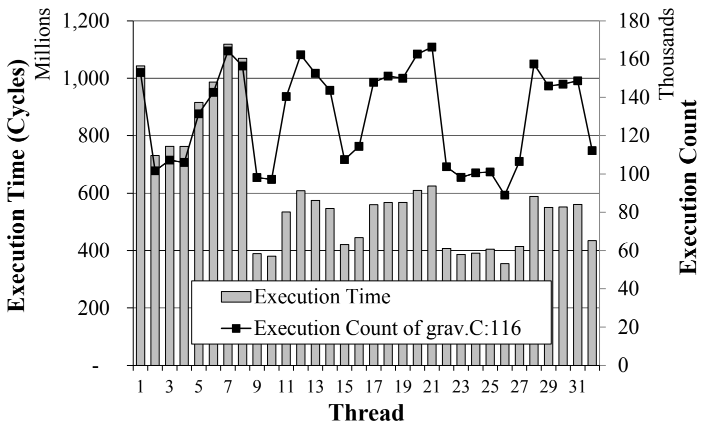
4.2 LIME Analysis Extension

LIME extension for heterogeneous processors adjusts thread execution times to level event costs from different types of cores. The adjustment includes scaling up the execution times from the faster core sets to match those from the slowest core sets. In this study, the execution times of big cores are scaled up.

One important question here is how much to scale up. If scaling up too much, the big core's cost of events will be over-estimated, resulting inaccurate LIME analysis. For example, in Figure 14b, the cost of event *grav.C:116* in big core is more than



(a) Under-scaled execution times



(b) Over-scaled execution times

Figure 14: Examples of improper adjustment of execution times.

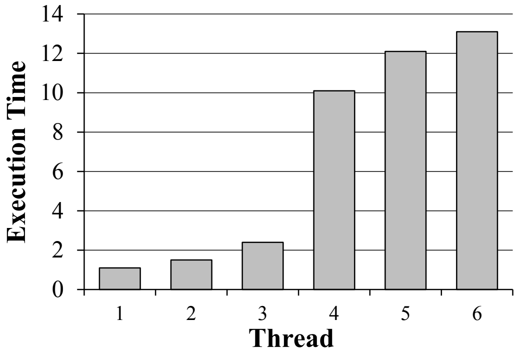
50% higher than that of small cores. Here, the correlation coefficient between event counts and execution time drops to 53%, compared to 99.9% with homogeneous cores. Similarly, not scaling enough causes under-estimated event cost at the big core and degrades analysis accuracy (Figure 14a).

4.2.1 Scaling Thread Execution Time

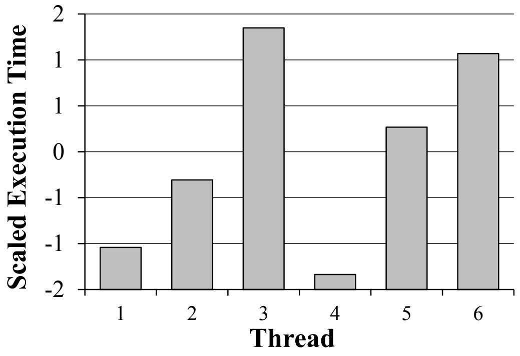
Here, I present 4 ways to estimate how much to scale up the execution times. Note that some methods work fine for a set of applications but fail on the other applications. For those that do not perform well, I present potential reasons for failure and ways to improve.

Table 9: Example of execution time scaling using standard scores.

	Big Core			**Small Core**		
	Thread 1	Thread 2	Thread 3	Thread 4	Thread 5	Thread 6
Execution Time	1.10	1.50	2.40	10.10	12.10	13.10
Big core Standard Score	-1.04	-0.31	1.35			
Small core Standard Score				-1.34	0.27	1.07
Scaled Execution Time	-1.04	-0.31	1.35	-1.34	0.27	1.07



(a) Original execution time



(b) Scaled execution time

Figure 15: Execution time before and after scaling using standard scores.

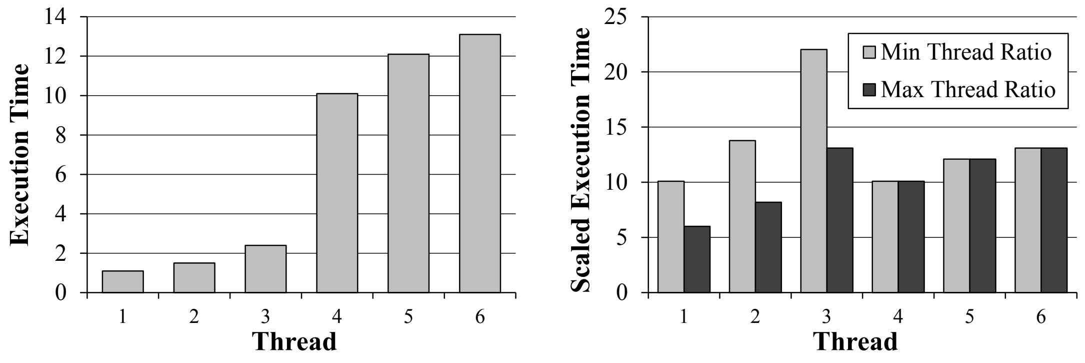
4.2.1.1 Standard Score

The first method is standard score [52], or Z-score. Standard score is also used in LIME analysis process to combine multiple event counts in a cluster regardless of their absolute values. To exploit the standard score in the execution time scaling, I first convert execution time from both big and small cores to the standard score separately. Then, the two sets of standard scores are concatenated to form new adjusted execution times.

Table 9 demonstrates how to calculate standard scores and concatenates them. The table displays 6 cores with 3 big cores (1 to 3) and 3 small cores (4 to 6) with the execution time of 6 threads run on each core. Note that in the second and the third row of the table, the standard score of big core and small core execution time are calculated separately. This allows us to convert them to have the same mean (0) and the same standard deviation (1). After the conversion, it is possible to compare them based on the same criteria. Figure 15 shows both the original execution time and scaled execution time. Among the big cores, thread 3 runs slowest but still the

Table 10: Example of execution time scaling using standard scores.

	Big Core			Small Core		
	Thread 1	Thread 2	Thread 3	Thread 4	Thread 5	Thread 6
Execution Time	1.10	1.50	2.40	10.10	12.10	13.10
Min Thread Ratio	10.10	13.77	22.04	10.10	12.10	13.10
Max Thread Ratio	6.00	8.19	13.10	10.10	12.10	13.10



(a) Original execution time

(b) Scaled execution time

Figure 16: Execution time before and after scaling using min/max thread ratio.

execution time is much smaller than the small cores due to its superior performance. Standard score adjusts the execution times of both big and small cores and reveals relative time difference between threads within each core group.

This standard score works well with applications in which thread behaviors do not show much variance between different types of cores. The problem is that if threads in one core group execute far more events than the others, separate scoring for each core group results in inaccurate shift of execution times. This is because the score does not consider threads' behavior in its scaling. For example, let's assume thread 1, 2 and 3 execute twice as many events than before. The original execution time in Figure 15a will be doubled (roughly), but the scaled execution time remains unchanged as we "separately" scale big and small core execution times. This oblivious scaling will break LIME analysis and deteriorate accuracy of the results.

4.2.1.2 Min/Max Thread Ratio

Min/Max thread ratio tries to overcome the shortfall of the standard score by scaling execution times using ratio between the minimum (or maximum) execution times of big and small cores. In the example above, the minimum execution time, i.e., the runtime of the fastest thread is 1.10 on the big cores and 10.10 on the small cores. The ratio between these two becomes $\frac{10.10}{1.10} = 9.18$. I scale execution times of threads on big cores by multiplying this ratio as shown in the second row of Table 10. Max thread ratio is similar to the min ratio, but the factor is calculated using maximum execution time of both big and small cores. As shown in Figure 16, min ratio scaling equalizes the minimum execution time of both big and small cores while max ratio scaling equalizes the maximum execution time.

The ratio based scaling will restore the cost of events of the big core to the same level as the small one if the events executed by the threads with minimum (or maximum) execution time are similar. However, some applications execute different types or numbers of events for different types of cores. For those applications, this ratio based scaling fails.

4.2.1.3 Max Thread Cost

To overcome the shortfall of min/max thread ratio scaling, I explicitly calculate the average event cost of the latest thread with maximum execution time for both big and small cores. For thread x, the average event cost c_x, called max thread cost, is $c_x = \frac{T_x}{\sum_{e \in E} C_{x,e}}$ where T_x is the execution time of thread x, E is the set of all events and $C_{x,e}$ is the count of event e for thread x. Now, if thread x is the latest thread on the big cores and y is the latest on the small cores, scaling factor using max thread cost becomes $\frac{c_y}{c_x} = \frac{T_y \cdot \sum_{e \in E} C_{x,e}}{T_x \cdot \sum_{e \in E} C_{y,e}}$. I multiply this scaling factor to the thread execution time of the big cores.

This max thread cost can estimate the performance difference between two types of cores better than min/max thread ratio. This is because it uses per-thread cost that is not affected by the number of executed events in the latest threads of big and small cores. Unlike this, min/max thread ratio is largely dependent on the absolute number of events each thread executes, and fails to provide accurate estimation if either one of them executes far more events.

4.2.1.4 Average Thread Cost

Scaling using average thread cost tries to improve accuracy further by taking all events from all threads into account for event cost estimation. Max thread cost method fails when the latest threads from big and small cores execute different types of events. If the latest threads follow different execution paths, the profiled events will have little overlap and the event costs for the big and small cores will be estimated on different bases. For example, in *radiosity* from SPLASH-2, the latest thread among the big cores follows an almost completely unique execution path, so only 0.3% of the events are executed by the latest thread from small cores.

To overcome the problem of disjoint events, in this average thread cost method, I increased the candidate event pool by including all events from all threads in the event cost calculation. Specifically, the new event cost c_b for n big cores is now $c_b = \frac{\sum_{x=1}^{n} T_x}{\sum_{x=1}^{n} \sum_{e \in E} C_{x,e}}$, where T_x is the execution time of thread x, E is the set of all events and $C_{x,e}$ is the count of event e for thread x. Here, the scaling ratio $\frac{c_s}{c_b}$ is multiplied to the execution time of the big cores, where c_b is the cost for big cores and c_s is the cost for small cores. By considering all events from all threads, more events are covered in the estimation process. In the *radiosity* example, the percentage of events commonly observed in both big and small cores increases from 0.3% to 5.4% with the new averaging method.

4.3 Experiments and Results

In this section, I present the experiment setup and results from the various scaling methods described in the previous section.

Table 11: Configuration of 32-core heterogeneous processor with 8 big cores and 24 small, throughput cores.

	Small Cores	**Big cores**
Core		
Number of Cores	24	8
Frequency	800 MHz	2.96 GHz
Instruction Queue	20	28
Decode/Rename/Wakeup	2/1/2 cycles	3/2/3 cycles
Store-Forward Delay	2 cycles	1 cycle
ROB Size	64	168
Integer Register	128	160
FP Register	64	144
Integer Unit		
Number of Units	8	3
Window Size	20	54
Scheduling Port	2	3
ALU/Div/Mult Latency	1/8/1 cycles	1/20/4 cycles
Floating Point Unit		
Number of Units	12	3
Window Size	54	32
Scheduling Port	8	3
ALU/Div/Mult Latency	1/1/2 cycles	1/4/24 cycles
Memory System		
L1 Cache	16 KB I/D, 8 way, 2 cycles	32 KB/64 KB I/D, 8 way, 2 cycles
L2 Cache	32MB, 16 way, 20 cycles	
Memory	323 cycles	

4.3.1 Implementation and Experimental Setup

The extension I present in this chapter is implemented based on the LIME framework. LIME analyzes the adjusted execution times along with profiled data to detect imbalance-related performance bugs. The profilers presented in LIME (Section 3.3) and analysis routine remain unchanged. Scaling is implemented in C++ and executed before the LIME framework.

The heterogeneous extension is tested using the same applications experimented with LIME. 14 parallel applications from SPLASH-2 and PARSEC benchmark suites are simulated at heterogeneous processor. SESC simulator [79] was modified for multiple cores to run with different clock frequencies. The used configuration is described in Table 11. This study tested 32-core CMP with a fixed number of small and big cores. 8 big cores are designed for better single thread performance and employ high clock frequency (2.96 GHz) with large reorder buffer (ROB) and registers. These big cores also exploit larger private instruction and data caches. Additionally,

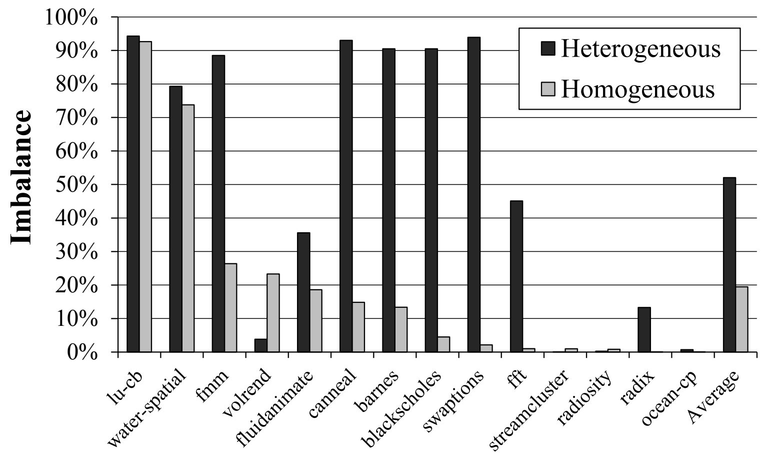


Figure 17: Exacerbated load imbalance on heterogeneous cores due to the performance difference.

24 small cores have been designed for better efficiency and then packed for high throughput. Small cores run slower than big cores with 800 MHz clock frequency and smaller ROB/registers. Each type of cores is backed by private L1 instruction and data caches that share 32 MB L2 unified cache.

4.3.2 Heterogeneity and Load Imbalance

We first present how the load imbalance changes as we employ heterogeneous cores compared to the homogeneous case. Figure 17 shows the percentage of load imbalance when applications are simulated with the heterogeneous configuration. Note that the applications are not yet optimized for the heterogeneous cores. In general, heterogeneous cores exacerbate the load imbalance. This is because threads on the big cores complete assigned jobs fast and reach synchronization point earlier than those on the small cores. Those faster threads wait for the slow threads on small cores, leading to higher load imbalance. *volrend* is the only application that shows reduced load imbalance with heterogeneous cores. In *volrend*, imbalance was caused by the main thread that handles output file. This main thread was executed by a big core, which shortened the overall execution time and reduced load imbalance.

Note that workload allocation did not occur according to the cores' performance

Table 12: Scaling ratio with various scaling method. The ratio denotes how slow the small cores are than the big cores.

Benchmark	**Std Score**	**Min**	**Max**	**Max-Cost**	**Avg-Cost**
radiosity	1.8	5.4	0.5	142.7	61.5
volrend	0.0	33.9	0.0	32.9	25.6
lu-cb	265.1	18.9	799.5	20.2	20.6
radix	19.3	19.5	19.0	19.0	19.3
ocean-cp	17.4	17.4	17.4	17.4	17.4
streamcluster	0.1	39.9	0.0	21.7	17.0
fluidanimate	13.2	11.4	17.1	17.1	16.2
swaptions	16.1	16.0	16.1	16.2	16.1
water-spatial	13.2	7.9	21.2	15.6	15.8
fft	14.9	14.9	14.7	14.7	14.9
barnes	13.4	12.2	13.9	13.8	13.7
fmm	12.6	13.2	12.3	56.3	12.8
blackscholes	10.5	10.5	10.5	10.5	10.5
canneal	10.5	10.6	10.0	9.7	10.5
Average	29.1	16.5	68.0	29.1	19.4

in this extension. Previous studies [64] have shown that intelligently allocating more jobs to faster thread improves parallel performance of applications by reducing the load imbalance caused by the heterogeneity-agnostic workload assignment. However, as denoted in Section 4.1, this study focuses on detecting source of the remaining, program-intrinsic load imbalance.

4.3.3 Scaling Results

Table 12 shows the results of various scaling methods presented in the previous section. The table shows how slow the small cores are compared to the big cores. For example, if the ratio is 10.0, it means that on average, the small core takes 10 times longer than the big core to complete the same amount of work.

Scaling using standard score shows reasonable estimation of scaling factor for some applications. However, it fails to provide meaningful number for many applications including *volrend*, *radiosity* and *streamcluster*. Those applications show better results with the min or max scaling. However, it is likely that only one of them succeeds. For example, in *lu-cb*, min scaling gives reasonable output of 18.9 but max scaling produces a value of 799.5. This is because in many cases, either the latest or the fastest thread follows the same path in both big and small cores. Consequently, min scaling tends to fail when max scaling succeeds, and vice versa. Max thread cost (Max-cost) alleviates this problem by calculating per-event costs. As shown in the table, max thread cost produces reasonable output for the most applications except for *radiosity*. Max thread cost method on *radiosity* suffers from non-overlapping events from the latest threads of the big and small cores. By considering all threads in the event cost calculation, average thread cost mitigates the inaccuracy and lowers the ratio to a reasonable level. With average thread cost, LIME could report the source of imbalance for *radiosity* but max thread cost scaling failed to produce output with LIME.

Note that the methods presented in this thesis is not a complete list. In fact, any scaling ratios will work with LIME extension as long as they provide reasonable estimate on the performance difference between the big and small cores. For example, one can apply a constant ratio that is based on some combination of hardware-related attributes to all applications. In this thesis, I use two constant ratios based on ROB size, cache size and clock frequency.

$$\left\lceil \ln \frac{\text{Big core's ROB size}}{\text{Small core's ROB size}} \right\rceil \times \left\lceil \ln \frac{\text{Big core's cache size}}{\text{Small core's cache size}} \right\rceil \times \frac{\text{Big core's clock freq.}}{\text{Small core's clock freq.}} \quad (3)$$

$$\frac{\text{Big core's ROB size}}{\text{Small core's ROB size}} \times \ln \frac{\text{Big core's cache size}}{\text{Small core's cache size}} \times \frac{\text{Big core's clock freq.}}{\text{Small core's clock freq.}} \quad (4)$$

Equation 3 and Equation 4 describe how I calculate those two constants. With the heterogeneous configuration used in this study, constant 1 from the first equation is 7.4 and constant 2 from the second equation is 13.5. I present how LIME output changes when the profiled data (big cores' execution time) is adjusted using these constants.

The scaling ratio can also be used to optimize applications for heterogeneous cores. One important procedure of the optimization process is to estimate how much more work we should assign to the big cores compared to the small cores. The scaling ratio can act as a proxy for the estimation because it tells us an approximated performance

Table 13: Event costs calculated using average thread cost.

Benchmark	**Average Event Cost**	
	Big Core	Small Core
blackscholes	20.6	215.6
fluidanimate	10.1	163.8
volrend	7.5	191.2
streamcluster	10.7	180.9
lu_cb	13.0	247.7
barnes	21.8	298.3
radiosity	10.1	611.9
fft	27.4	407.1
radix	17.6	339.1
water_spatial	12.4	196.1
swaptions	17.1	275.3
canneal	22.8	166.0
ocean_cp	18.0	313.0
fmm	38.3	488.8
Average	17.7	292.5

difference between the big and small cores. Additionally, one can verify whether the heterogeneous optimization is adequate by comparing the actual distribution of workloads with the scaling ratio. If they deviate much, the report will identify the workload assignment between the big and small cores as the culprit of the resulting load imbalance caused by the heterogeneous cores.

Table 13 lists the estimated event cost by using the average thread cost (Avg-cost). This cost was considered as an average cycle each thread takes to execute a control-flow event, i.e., cycles between two consecutive branches. The cost ranges from 7.5 (*volrend*) to 38.3 (*fmm*) for the big cores and from 163.8 (*fluidanimate*) to 611.9 (*fmm*) for the small cores. As a result, the scaling ratio varies between 10.5 and 25.6 (except *radiosity* where empty while loops distort the average event cost). Note that the clock frequency of a big core is 3.7 times higher than the frequency of a small core, and the private cache size of the big core is 4× (2× for I-cache) of the small core's cache. This emphasizes that simple factors like clock frequency of cache size are insufficient to represent the core performance, and one has to consider multiple factors at the same time to draw more accurate comparison between cores. The results in this study show that the average thread cost can be a viable proxy that is suitable for performance comparison among heterogeneous cores.

Table 14: Scores reported by LIME analysis for homogeneous cores (Section 3.4.2) and for heterogeneous cores. For heterogeneous case, big cores' thread execution times are scaled using average thread cost.

Benchmark	**Control Flow**								**Cache Miss**			
	Homogeneous		Avg. event cost		Constant 1		Constant 2		Homogeneous		Avg. event cost	
	Report	Score	Report	Score	Report	Score	Report	Score	Report	Score	Report	Score
lu-cb	1 (5)	**1.00**	1 (1)	**1.00**	1 (1)	**0.84**	1 (1)	**0.98**	0 (0)	–	0 (0)	–
volrend	1 (1)	**1.00**	1 (1)	**1.00**	1 (1)	**1.00**	1 (1)	**1.00**	0 (0)	–	0 (0)	–
fmm	2 (3)	**0.41**	1 (5)	**0.34**	0 (1)	0.08	2 (9)	**0.27**	0 (4)	0.03	2 (25)	0.16
barnes	1 (6)	**0.93**	1 (6)	**0.94**	1 (1)	0.26	1 (5)	**0.93**	0 (0)	–	0 (0)	–
canneal	2 (3)	**0.77**	2 (4)	**0.73**	0 (1)	0.00	2 (3)	**0.58**	0 (9)	0.06	0 (5)	0.08
fluidanimate	1 (2)	**0.13**	1 (1)	**0.13**	0 (0)	–	1 (1)	**0.13**	0 (0)	–	0 (2)	0.05
water-spatial	2 (2)	**0.62**	2 (2)	**0.62**	1 (2)	**0.52**	2 (2)	**0.61**	0 (1)	0.07	0 (5)	0.05
streamcluster	2 (2)	**0.75**	2 (2)	**0.75**	2 (2)	**0.75**	2 (2)	**0.75**	0 (0)	–	0 (0)	–
radiosity	2 (2)	**0.99**	3 (6)	**0.90**	2 (5)	**0.96**	3 (5)	**0.91**	0 (2)	0.03	0 (2)	0.02
swaptions	6 (6)	**1.01**	6 (6)	**1.00**	1 (3)	**0.91**	1 (3)	**0.91**	0 (5)	0.06	0 (0)	–
blackscholes	1 (1)	0.32	3 (5)	0.32	0 (1)	0.02	0 (0)	–	4 (7)	**0.63**	9 (10)	**0.61**
fft	0 (0)	–	3 (4)	**0.19**	0 (0)	–	0 (0)	–	0 (0)	–	0 (0)	–
radix	0 (0)	–	0 (2)	0.07	0 (0)	–	0 (0)	–	0 (2)	0.04	3 (3)	**0.35**
ocean-cp	1 (1)	**0.31**	1 (1)	**0.12**	0 (0)	–	0 (0)	–	0 (2)	0.05	0 (2)	0.03

4.3.4 Heterogeneous LIME Results

To verify the estimated scaling factors between big and small cores, LIME analysis was applied with the scaled data. Reasonably estimated factor will scale up the thread execution times of big cores so that the LIME framework can process them as if they are profiled from homogeneous cores.

Table 14 lists the result of LIME that has the event data profiled on both homogeneous and heterogeneous cores. Heterogeneous data are adjusted using three methods: average event cost (Section 4.2.1.4), Equation 3 (*constant 1*) and Equation 4 (*constant 2*). As shown in the table, LIME extension with scaling based on average thread cost allows LIME to detect and report similar scores for the heterogeneous data with the homogeneous case. The manual inspection of the reported code lines verified that they point exactly the same locations for applications with the control-flow caused imbalances. When a constant ratio is used to adjust the execution time, LIME sometimes fails to produce correct output if the constant is too far from the actual ratio. For example, LIME produces incorrect results for 4 applications as the value of constant 1 deviates more from the reported event cost ratios (7.5 vs. between 10 and 20). LIME reports correct bug points with the second constant (13.5) that is close to the measured ratio.

For applications that show imbalance caused by cache misses, heterogeneous extension successfully identified the cause for load imbalance in *blacksholes.* Note that *blacksholes* is the only application that LIME detects reasonable cause of load imbalance with the homogeneous case (Figure 5). It shows that the heterogeneous extension of LIME presented in this chapter can accurately measure the performance difference between the big and small cores and adjust profiled data, allowing the LIME framework to be applied to the heterogeneous processors.

CHAPTER V

SUPPORTING MULTIPLE FAST BARRIERS USING ON-CHIP TRANSMISSION LINES

In this chapter, I present TLSync, a support for multiple fast barriers using on-chip transmission lines that are designed to help alleviate barrier latency problem.

5.1 Design of TLSync

I propose a novel hardware barrier support which uses high-frequency RF signals for barriers, while still leaving the transmission line (TL) usable for baseband and low-frequency RF data transmission. This barrier, which is called TLSync, supports multiple simultaneous barriers by allocating a separate RF frequency band to each barrier, and uses very simple modulation to minimize cost and demodulation latencies. Finally, TLSync accommodates the fact that weakening and distortion in TLs get worse as the frequency of the signals grows—the high-frequency RF bands I use are unlikely to be of practical use for data transmission, but are good enough for the simpler signals used for TLSync barriers.

TLSync barrier support is conceptually very similar to traditional wired-OR [4], wired-NOR [87], and wired-AND approaches, which differ mainly in the interpretation of voltage levels (high/low vs. 1/0). A traditional wired-AND barrier would have a single chip-spanning wire, which would be connected to the supply voltage (logical 1) with a pull-up resistor and to each core by pull-down drivers. These drivers keep the wire at low (logical 0) voltage in spite of the current passing through the pull-up resistor. A core that has not yet arrived to the barrier would keep its pull-down driver active, thus keeping the wire at a low voltage level. When the core reaches the barrier, it simply deactivates its pull-down driver and starts monitoring the wire. When the last core has arrived, no pull-down drivers remain active and the pull-up resistor brings the wire's potential to a high value, which signals to all cores that the barrier is complete.

Unfortunately, an actual chip-spanning wired-AND circuit would have a huge latency because the wire has a large capacitance. For example, in the 45nm technology, a wired-and circuit for 16 4mm $\times$ 4mm cores (for a total chip size of 16mm $\times$ 16mm) would have a 34.7ns delay[1]—nearly 100 cycles for a 3GHz core. This latency grows quickly as technology scales—in 32nm technology, 32 such cores would fit on the same die area and the chip-spanning wired-AND would have a 100ns delay[2], and in 22nm technology the wired-OR for 64 cores would have a 312ns latency [3].

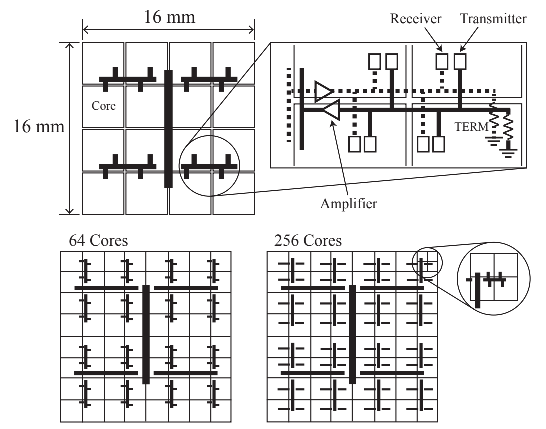
The latency of a traditional wired-AND circuit can be improved using a reduction tree for the AND operation and then a notification tree to distribute the final result to all the cores. With this approach, the chip-spanning barrier network would have 8.7ns, 23.4ns, and 59.4ns latency for the 45nm/16-core, 32nm/32-core, and 22nm/64-core chips used in the previous example. This latency can be further improved by using optimal repeater spacing along each wire segment—the delay drops to only 1.3ns, 2.4ns, and 3.4ns for the same three chips, but at a cost of using silicon (for repeaters) at frequent intervals along the wire's path. This use of silicon also interferes with the layout of the devices (cores, caches, etc.) along the path of the wire.

Even with the repeated-wire approach, the technology scaling trend is unfavorable and each reduction/notification barrier tree only provides support for a single barrier (i.e., only one parallel application). If there are multiple groups of threads that synchronize on independent barriers (e.g., several multi-core applications running

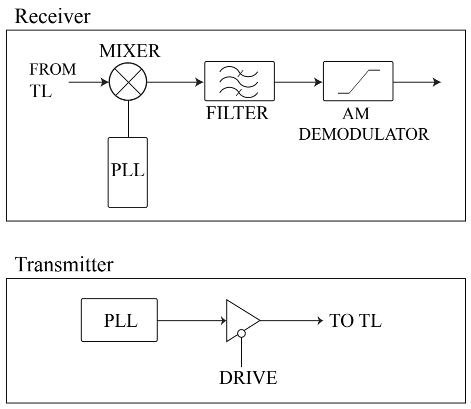
[1]The chip-spanning wire would have a total length of 64mm (16mm$\times$4) to connect all 16 cores, while ITRS [43] reports a 0.54ns/mm latency for an unrepeated (continuous) wire in 45nm technology.

[2]Total wire length grows to 88mm and wire delay is expected to be 1.13ns/mm in 32nm technology

[3]Total wire length of 126mm, with 2.48ns/mm delay



(a) Chip-Spanning Transmission-Line Network



(b) Receiver/Transmitter block diagram

Figure 18: Overall design of the TL network and TLSync transmitters and receivers.

concurrently), separate hardware barrier support would be needed for each group.

TLSync barrier retains the conceptual simplicity of wired-AND circuits and their main advantage—there is no contention between arriving cores, so when the last core arrives to the barrier, all cores are released after a fixed latency, even if all cores arrive nearly simultaneously. However, TLSync eliminates the problems of latency scaling (by using transmission lines) and of supporting multiple groups (by using different frequency bands for different groups).

Logical zero and one in a TLSync barrier are represented by the presence or absence of a signal at a particular frequency (the simplest form of binary amplitude modulation). Each group of cores that uses barrier synchronization is dynamically assigned to a particular frequency, thus separating it from other groups without requiring a separate set of transmission-lines (TLs) for each group. Each core has a transmitter and a receiver (Figure 18b), both tuned to the frequency assigned to the group the core belongs to. The transmitter outputs a continuous signal at this frequency into the TL network, and the receiver detects the presence of such a signal. When a core arrives to the barrier, it stops transmitting. When the last core in a barrier-synchronized group has arrived and stops transmitting, after a short delay (propagation and receiver delay) all the cores in the group detect the absence of the signal and can continue past the barrier. Because different groups of cores can use different frequencies, many barriers can be implemented simultaneously using the same TL network and the same transmitter/receiver circuitry in each core. The only change from using a single, fixed, frequency for all cores is to make the transmitter and receiver tunable (see Section 5.2).

Note that the design of a TL based broadcast network is not a straight-forward task. The number of cores can be large and a single TL segment can have a limited number of connections (see Section 5.2). To overcome this problem, I use multiple TL segments connected using amplifiers instead of routers and repeaters, so all of the segments still form the same broadcast medium (see Figure 18a). Section 5.2 also provides implementation details for the TL broadcast network and for transmitters and receivers used in the implementation of TLSync barriers.

The total delay of a TLSync barrier is composed of three types of latencies: 1) the TL propagation time, which largely depends on the total distance traveled between the farthest-apart transmitter and receiver, 2) transistor circuit latencies, which depend on the complexity of the circuitry in transmitters, amplifiers, and receivers, but continues to improve with technology scaling, and 3) delays introduced by fundamental requirements of signal processing in filters and demodulators. I now summarize the various components of TLSync barrier latency, in the order in which these delays are encountered by the transmitted signal.

The transmitter delay contains only a single CMOS pass-gate, which directly connects the output of the digital phase-locked loop (PLL) to the transmission line (see Figure 18b). This delay is only a few picoseconds and improves with technology scaling. Propagation speed along transmission lines is nearly constant at 0.0075ns/mm [66] and is largely unchanged by technology scaling. For a 16mm$\times$16mm die with 16 cores, the worst-case length of TL segments traversed by a signal in this broadcast network is 24mm and there are three amplifiers along the way—as shown in Figure 18a, there is a local *collection* segment (4mm) with four transmitters, then an amplifier feeds the collected signals to the central collection segment (8mm). The overall collected signal is amplified and sent onto the central distribution segment (8mm again), then amplified again and sent to the local distribution segment (4mm again). Each amplifier introduces 0.05ns of delay [88], for a total propagation delay (transmitter output to receiver output) of 0.33ns. When the number of cores is increased to 64 (in 22nm), the longest path is 32mm (8mm, 6mm, and 2mm central, quadrant, and local segments for collection and for distribution). There are now five amplifiers along each path, but their latency has improved with technology [88] to 0.025ns, so the total propagation latency is 0.365ns. With 256 cores in 10nm technology, the path is 36mm with seven (even faster) amplifiers, for a total propagation latency of 0.358ns (36mm $\times$ 0.0075ns/mm + 7 amplifiers $\times$ 0.0125ns/amplifier).

At the receiver, a mixer in 45nm technology introduces a 0.5ns delay [2], which is expected to improve to 0.25ns and 0.13ns with technology scaling to 22nm and 10nm. Finally, band-pass filter and amplitude tracker (AM demodulator) latency is dominated by fundamental signal processing delays. The demodulator's delay depends on the frequency of its input signal (filtered mixer's output). I use a 1.8GHz signal, resulting in a 0.28ns demodulator latency (half of the period). The filter creates a fundamental tradeoff between latency and frequency band allocation[4]. When one allocate to a barrier larger spectrum around its "tone" frequency, it achieves faster filtering but reduce the number of barrier groups that can be supported. I experimented with frequency bands as narrow as 100MHz to as wide as 500MHz, designing the filter to have a frequency profile just steep enough to guarantee that signals at

[4]The fundamental filter delay constraint is a close cousin of the sampling theorem—to preserve a signal at one frequency while suppressing another whose frequency differs by x GHz, the filter must "observe" the signals for $1/x$ ns. Actual filters have small additional circuit delays and can save some time by having less suppression for "neighboring" signals, but their delay is still primarily determined by this fundamental constraint.

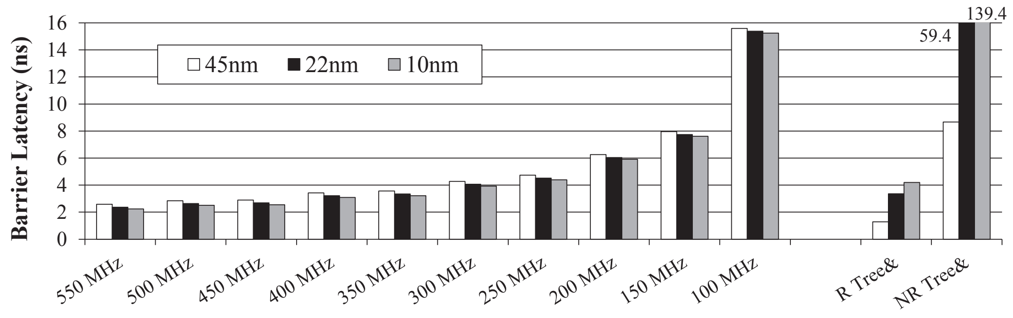


Figure 19: TLSync Barrier latency for different frequency band allocations.

surrounding frequencies will not result in false positives of false negatives for a barrier. Filter designs that worked well in most cases were 3-stage digital infinite impulse response (IIR) filters, with delays of 1.74ns, 2.32ns, 3.17ns, 5.15ns, and 14.48ns when using 500MHz, 400MHz, 300MHz, 200MHz, and 100MHz bands, respectively [66][5].

The total end-to-end delay for the barrier implementation, from the time the last core arrives to the barrier to the time all processors are notified, is shown in Figure 19 for different frequency band allocations and technology nodes. This delay includes propagation and receiver (mixer, filter, demodulator) delays. For comparison, I also show latencies for a traditional-wire reduction-notification tree barrier with optimal repeater spacing and sizing ("R Tree&") and without repeaters ("NR Tree&"). Recall that a traditional-wire tree can only support one barrier, and that optimally-repeated wires rely on using silicon along the wire's path.

I observe that, at the 45nm node, this barrier implementation has a latency of 2.85ns with a 500MHz band assigned to each group. Note that with 16 cores there are at most eight 2-core groups, so 500MHz per group still leaves more than 5GHz of spectrum for other uses. However, the optimally-repeated traditional barrier tree has a latency of only 1.28ns, and with only 16 cores only a few "R Tree&" networks may be needed on a chip.

[5]The actual filter design was for 450MHz, 350MHz, etc. pass-band, leaving 25MHz on each side of the pass-band to reduce interference from neighboring signals in order to avoid use of even more filter stages.

However, at future technology nodes the latency of a traditional barrier network sharply increases while the number of cores (and thus potential barrier groups) also increases. In contrast, the latency of the TLSync barrier implementation improves slightly, as faster transistor circuits compensate for longer paths traversed by the signals. More importantly, the same TL network can support multiple (many) simultaneous barriers that can be active when many cores are available.

I also observe that, regardless of technology node, filter delays result in significant barrier latency increase when the frequency band allocated to a synchronization group is 200MHz or smaller, and that other latency components are dominant when the allocation is more than 450MHz per group. Between 250MHz and 450MHz per group, there is a reasonable tradeoff between spectrum consumption and barrier latency. One way to manage this tradeoff is to vary size of the band at runtime, depending on how many barriers are active. With fewer than 10 barrier-synchronized thread groups, each group would get a 500MHz band to support a 2.64ns barrier latency at 22nm and 2.50ns latency at 10nm. With more groups, progressively smaller bands would be allocated to each group, with a small gradual increase in barrier latency experienced by each group.

5.2 Implementation Details

5.2.1 Using TLs to Connect Many Cores

TLSync mechanism relies on a chip-wide broadcast network built with transmission lines (TLs). Ideally, this broadcast would be provided by connecting all transmitters and receivers to the same TL (that winds its way through the chip). However, each such connection changes the electromagnetic field that propagates along the TL, thus weakening and distorting the signals represented by this field. Detailed simulations of electromagnetic propagation in transmission lines (see Section 5.2.3 for details) show that, for on-chip distances, the length of the TL segment is almost irrelevant—longer segments do have more signal attenuation (weakening) and reflection (which causes distortion), but the main factor that determines the overall attenuation and distortion is the number of connections.

To overcome this problem, I use multiple TL segments instead of a single TL (see Figure 18a). These segments are connected using amplifiers instead of routers and repeaters, so all of the segments still form the same broadcast medium. In light of the results from the electromagnetic simulations (Section 5.2.3), I use TL segments with at most five connections. The simulations show that the attenuation of such a segment can still be compensated for by a relatively simple amplifier, and that reflected signals are still weak enough to not interfere much with transmitted signals.

When connecting multiple segments using amplifiers, one must avoid positive feedback loops. In practice, this means that I cannot connect a TL segment to another one using amplifiers in both directions, and I cannot form a ring of segments. Instead, the TL network I use consists of a quaternary tree of input segments, possibly a shared root segment, and a quaternary tree of output segments. Transmitters from four cores are connected to a leaf "collection" segment, which feeds an amplifier. The outputs of four such amplifiers are connected to a second-level collection segment, which feeds an amplifier connected to a third-level collection segment, etc. If the number of cores on chip is a power of four, this tree has a root collection segment that feeds an amplifier whose output is connected to the root distribution segment, which feeds four amplifiers connected to distribution segments, etc. until eventually each leaf distribution segment feeds four receivers (see Figure 18).

If the number of cores is not a power of four, the root collection and distribution TL segments each have fewer connections, so a single root segment can be used for both. For example, with 32 nodes, eight leaf collection segments feed two second-level collection segments, and two second-level distribution segments feed the eight leaf distribution segments. As a result, all four second-level segments (two for collection and two for distribution) can be connected using a single root segment.

By placing the root TL segments in the middle of the chip and by carefully laying out the others, the total length of TL that must be traversed by any signal (from a transmitting core to the farthest-away receiving one) is close to the sum of the width and length of the chip, plus a few millimeters of extra distance to avoid having turns in a transmission line segment—a segment with a turn would have additional attenuation and reflection, and would require a separate analysis to determine how many connections can be allowed.

5.2.2 Transmitter and Receiver Design

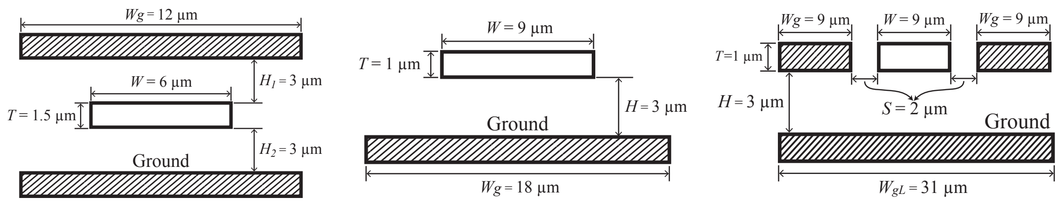
When designing transmitters and receivers for a multi-core broadcast network based on transmission lines (TLs), the main goal is to preserve signal fidelity without introducing a significant delay. This is not a simple task, because signals with different frequencies propagate differently in TLs—lower-frequency signals propagate less quickly than high-frequency ones, but high-frequency signals suffer more attenuation and more distortion due to crosstalk, skin effect, and other artifacts of electromagnetic wave propagation. These effects affect non-modulated high-data-rate signals: rapid transitions have very high-frequency components and are thus significantly distorted; this limits the data rate that can be achieved without sophisticated modulation and multiplexing methods. On the other hand, sophisticated modulations and multiplexing could introduce signal processing delays that are incompatible with on-chip latency requirements. In fact, most of the high-data-rate modulation and multiplexing schemes used in modern telecommunications rely on digital processing in the receiver, with milliseconds-long delays. Such delays are several orders of magnitude longer than those needed for on-chip communications.

As a result of these considerations, practical on-chip data broadcasts are likely to use simple signaling methods (e.g., baseband signaling without modulation) and have relatively low data rates. Therefore, a significant frequency range below the TL cut-off frequency (see Section 5.2.3) may be unused, because signal integrity at those frequencies is not sufficient for data transmission without sophisticated (and long-latency) modulation schemes. These frequencies can still be used for other mechanisms that require low-latency broadcasts, such as TLSync barrier synchronization.

In light of this discussion, TLSync uses the simplest form of binary amplitude modulation (i.e., the presence or absence of a signal at a particular frequency) to indicate a logical zero or one. Transmitter delay is very small—when a core reaches the barrier, it simply turns off the pass-gate that connects the output of the PLL to the transmission line. However, the receiver design is more challenging.

To implement different groups of cores that use different frequencies for barrier synchronization, the only change from using a single, fixed, frequency for all cores is to make the transmitter and receiver tunable. A naive approach to providing tunability would require a tunable PLL in the transmitter, and a tunable filter (demodulator) in the receiver, but a tunable digital filter would have significant additional delays. Fortunately, a viable receiver can be implemented using a fixed-frequency filter by leveraging the same tunable PLL used in the core's transmitter. For this, the received signal first goes into a *mixer*, which combines the signal with a reference signal from the PLL. The output of the mixer is a *down-converted* signal—signals from the input signal are represented in the output, but their frequencies are reduced by the frequency of the reference signal[6]. The output from the mixer is then fed to a band-pass filter whose frequency is constant. Tuning is achieved by controlling the frequency of the reference signal. For example, if the filter's pass band is centered at 1.8GHz and the group's assigned frequency is 8GHz, the reference signal would be generated at 6.2GHz. To generate reference signals for transmitters and receivers, I use digital

[6]The output of the mixer also contains up-converted signals, whose frequency is the sum of input and reference frequencies. These signals are eliminated by the filter that follows.



(a) Stripline (b) Microstrip (c) Coplanar

Figure 20: Realizations of transmission lines.

PLLs that can be implemented efficiently. Tuning is achieved by changing the PLL's divider and multiplier factors, without changing phase compensation and other hard-to-design circuitry of the PLL.

Because this TLSync barrier only requires detection of the presence or absence of a “tone” at a given frequency, the band-pass filter used for the barrier receiver needs not have a very steep frequency profile—it only needs to have low attenuation at ”tone” frequency of the band allocated to the group and large-enough attenuation for “neighboring” tones, but the attenuation profile for frequencies in-between need not be sharply defined. This allows this filter to be both less expensive and faster: as explained in Section 5.1, the delay introduced by the filter fundamentally depends on how sharply its frequency profile is defined. Finally, the output of the band-pass filter is then fed to an AM demodulator—a diode and a small capacitor that tracks the amplitude of a high-frequency signal. The delays in the filter and AM demodulator are heavily dependent on the width of the filter's pass-band and on the frequency at which the AM demodulator is operating. In the proposed system, I use 1.8GHz for AM demodulator's frequency and several hundred MHz for the filter's pass-band.

5.2.3 Transmission Line Design

A transmission line (TL) consists of a signal conductor (wire), surrounding dielectric, and one or more grounding conductors that run in parallel with the signal wire. There are three main realizations of TLs that are potentially suitable for on-chip implementation: *stripline*, *microstrip*, and *coplanar waveguide*, as shown in Figure 20.

In designing TLs, the important characteristics are attenuation (forward loss), reflection (return loss), and propagation delay at different frequencies. These characteristics are, in turn, affected by not only dimensions of the conductor, but also the thickness and type of dielectric that separates it from the ground plane, by the fact that the "ground plane" is of finite dimensions, and by the size, shape, and impedance of any connections, bends, etc. This is further complicated by the fact that attenuation, reflection, and delay of TLs are not constant across all frequency bands—typically, there is a range of frequencies within which the TL has acceptable characteristics. A TL behaves as a low-pass filter—the attenuation slowly worsens as the frequency of the signal increases up to some frequency (called the cut-off frequency), with rapidly worsening attenuation beyond the cut-off frequency. The end result of this is that any relatively accurate characterization of TLs requires an electromagnetic (EM) propagation simulator (ADS, MWO, Linpar, etc.).

I used MWO electromagnetic simulator [66] to determine appropriate on-chip implementations for all three realizations of TLs. In this analysis, I focused on achieving good transmission characteristics (attenuation, reflection, latency) in a relatively large spectrum, while minimizing the total area consumption in the affected metal layers. After careful consideration, I selected a 1–10GHz spectrum as the target. Although a similar-sized spectrum at higher frequencies (e.g., at 56–65GHz, which is often used for on-chip RF circuitry in solid-state literature) would result in smaller and somewhat faster amplifiers, at higher frequencies the attenuation in TLs is much higher. To compensate for this, one would need larger number of amplifiers and very sophisticated analog circuitry at transmitters and receivers, which will increase the delay. As a result, I chose to operate in the ¡10GHz part of the spectrum. The optimized dimensions for all three realizations of TLs in a 45nm process are shown in Figure 20. The smallest dimensions are obtained for stripline realization, which I choose for TLSync design.

As a result of all these considerations and limitations, on-chip TLs must be designed carefully, especially when the geometry of TLs is further complicated by having more than just a single receiver and a single transmitter at the endpoints of the line. I find that these considerations have a major effect on the characteristics of the TL. As an example, consider a straight 16mm run of stripline implemented as in Figure 20a. With a single transmitter (at one end) and a single receiver (at the other end) that are both impedance-matched to the TL in the 1–10GHz frequency range, forward loss (attenuation) is 5dB at 1GHz and 10dB at 10GHz, return loss (attenuation of the reflected signal) is 15dB at 1GHz and 10dB at 10GHz, and propagation delay is 110ps. Although forward loss is significant (the received signal retains only 10% of the transmitted power), the signal is still significantly above the noise level and can be amplified using a low-noise linear amplifier. The return loss and delay, however, are excellent: the reflected signal is weaker[7] than the one being sent, so reflection causes little interference, and the delay is much lower than that of a traditional 16mm wire implementation (even with optimal repeater spacing). However, if one has 15 additional transmitters (at 1mm intervals along the TL), the forward loss, return loss, and propagation delay for the transmitter at the end of the TL become 44dB, 7dB, and 139ps at 10GHz. This signal is unlikely to be received successfully—the forward loss suppresses the signal well into the noise level, and the reflected signal is nearly 10,000 times stronger than the weakened forward signal. This loss occurs because impedance cannot be matched equally well in a wide frequency range (more than a few GHz), resulting in a slight-to-moderate impedance mismatch at some frequencies. This mismatch adds significant additional attenuation and reflection.

A practical implementation must have relatively few receiver/transmitter connections on the same TL segment, with careful planning and design of those connections.

[7]At 10GHz and beyond, the reflected signal becomes too strong, which is part of the reason why the usable spectrum for this transmission line ends at 10GHz

Table 15: Simulation Setup

Parameters	Small-Core	Large-Core
Core Frequency, Issue Width, ROB Size	1 GHz, 2 (Out-of-Order), 64	3 GHz, 4 (Out-of-Order), 128
Cache Line Size	64 Byte	64 Byte
L1 Instruction Cache	32 KB, Private, 2 way, 1 cycle	32 KB, Private, 2 way, 1 cycle
L1 Data Cache	32 KB, Private, 2 way, 1 cycle	32 KB, Private, 4 way, 3 cycles
L2 Cache	16 MB, Shared, 16 way, 6 cycles	256 KB, Private, 8 way, 5 cycles
L3 Cache	–	32 MB, Shared, 16 way, 23 cycles
Memory Latency	110 cycles	323 cycles
TLSync Barrier Latency (350/300/200/150MHz)	4/5/6/9 cycles	12/13/16/25 cycles

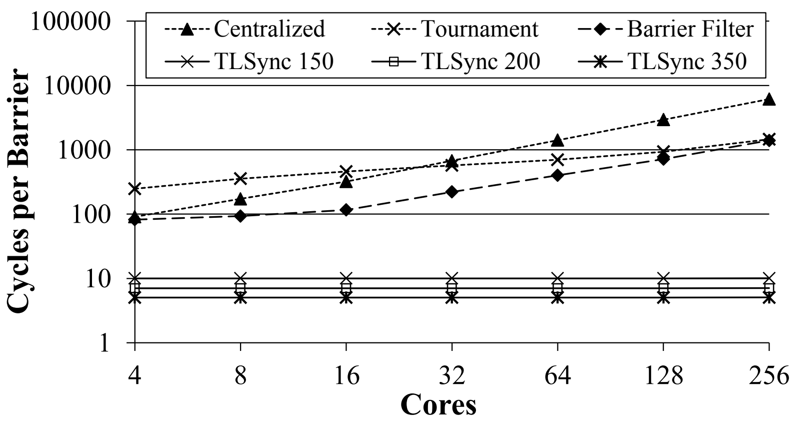
Through careful experimentation, modeling, and EM simulation, I found that up to 6-7 transmitters (and an amplifier at the receiving end) can be safely implemented on a single transmission segment. Reception segments are a bit more forgiving, allowing for 10-12 receivers (each with its own amplifier) to reside on a segment fed by an amplifier. TLSync design is conservatively engineered to have five connections (a transmitter and four receivers, or four transmitters and a receiver) on each segment.

5.3 Experimental Evaluation

I evaluate the performance impact of TLSync barrier in terms of raw barrier latency, and in terms of execution time on Livermore loop kernels [77], hand-parallelized Dithering application from EEMBC benchmark [29] and the Streamcluster application from the PARSEC benchmark [15]. Other PARSEC benchmarks use barriers rarely and their performance is not affected much by barrier latencies.

For comparison, I also show results for a optimized implementations of the widely-used centralized sense reversal software barrier (shown as "Centralized"), the tournament barrier (among the most scalable software barrier implementations [69]), and the hardware-assisted Barrier Filter [80] implementation. In all three implementations, I carefully arrange shared variables to avoid unnecessary coherence traffic and use the best-performing variant (e.g., ping-pong with I-Cache for Barrier Filter).

I obtain performance results using the SESC [79] cycle-accurate simulator. I model two types of cores, summarized in Table 15, with core counts from 4 to 256.



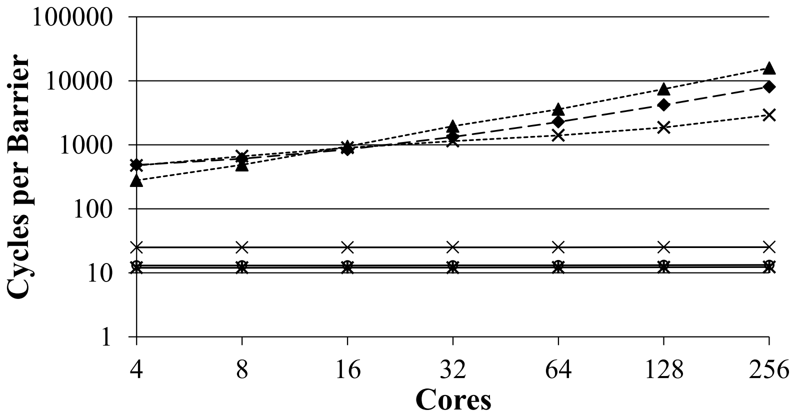


Figure 21: Raw barrier latency in Small-Core (top) and Large-Core (bottom) configurations.

The "Small-Core" setup models a "many-smaller-cores" approach that can achieve large core counts (64 and beyond) in the near future, while the "Large-Core" setup models the approach of using fewer high-performance cores, which can be expected to achieve large core counts later. The last row in Table 15 shows TLSync barrier latency with 350, 300, 200, and 150MHz bands allocated to each barrier in terms of clock cycles in each type of cores.

5.3.1 Barrier Latency

I measure the raw barrier latency by executing (in parallel) a tight loop with 64 barriers in the loop body, then dividing the execution time with the number of completed barriers.

Figure 21 shows the results from this experiment. In both configurations, TLSync barriers have very low latency which stays nearly constant as the number of cores increases. This is because the number of cores affects the barrier latency only slightly—e.g., from 16 to 256 cores (assuming the same 16mm by 16mm chip size) the longest path for signal propagation changes from 24mm to 36mm and there are four more amplifiers on that path. The resulting additional delay is only 290ps (less than one cycle even for "Large-Core"). The width of the frequency band allocated to the barrier changes its latency noticeably (note the logarithmic scale in Figure 21), but even with only a 150MHz band the TLSync barrier outperforms the others by an order of magnitude for small core counts and by two orders of magnitude for large core counts. As expected, the centralized sense reversal barrier outperforms the tournament barrier up to a certain point (16 small cores, 8 large cores). This is because the tournament barrier executes more code in each core but with little serialization among cores, whereas the centralized barrier uses a simple counter increment but serializes increments from different cores. Barrier Filter reduces barrier latency, but its scaling trend is similar to the other two because there is still serialization in the barrier filter logic that receives notifications from each arriving core.

5.3.2 Livermore Kernels

Livermore loops [77] have long been used in architecture and compiler research for testing the scaling of parallel performance. I use Livermore Kernels 2, 3 and 6 to evaluate the performance impact of different barrier implementations for workloads with large amounts of fine-grain parallelism. In parallelizing these kernels, I followed the same methodology that was used for Barrier Filter work [80], and I carefully aligned data structures to cache line boundaries and assigned data in cache-line-sized chunks to minimize coherence traffic. For the TLSync barrier, I only show the results when a 350MHz band is allocated to a barrier group, because results for bands sizes

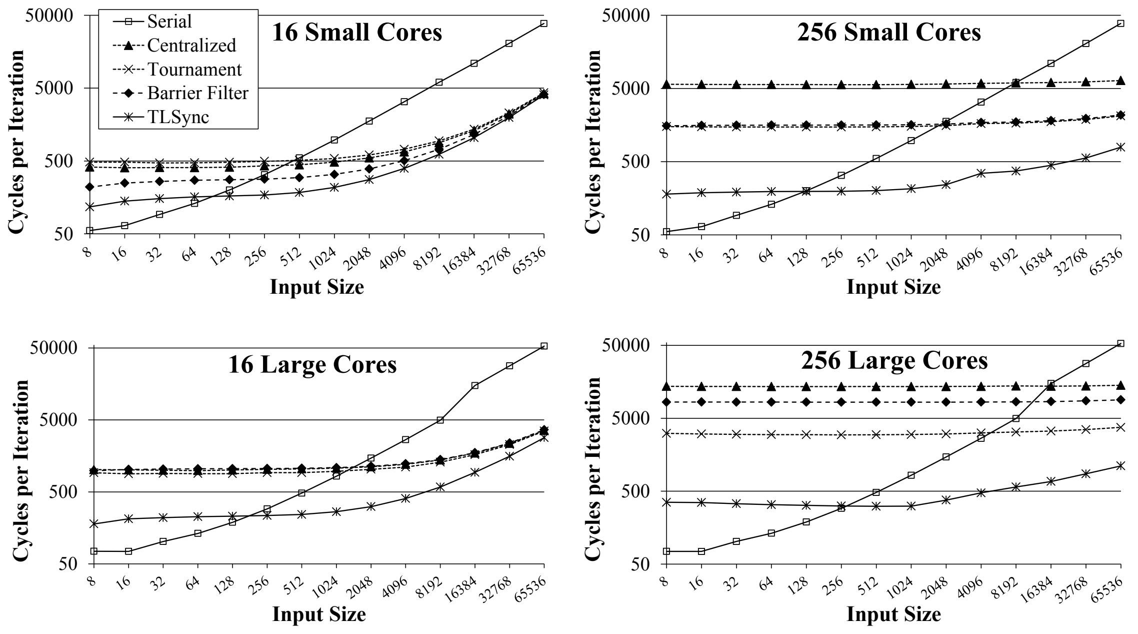
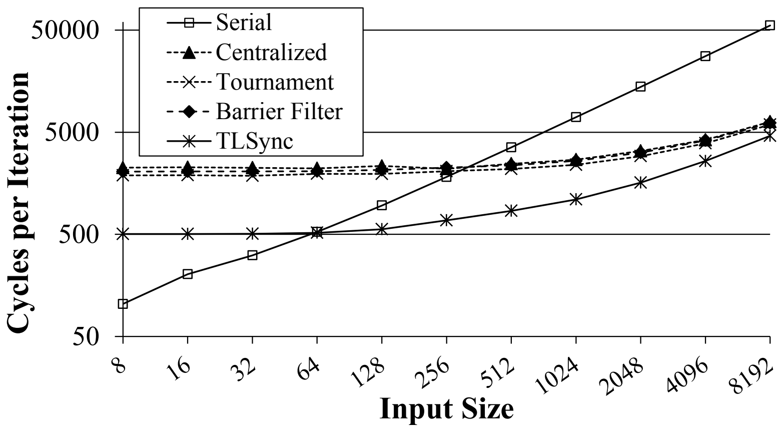


Figure 22: Performance of Livermore Loop 2 for various barrier mechanisms.

from 150MHz to 350MHz do not perceptibly differ from each other.

Figure 22 shows cycles per loop iteration in Kernel 2 for different barrier implementations across various input sizes. For comparison, each chart also shows the results for sequential execution ("Serial"). In each chart, as input size increases, more iterations are assigned to each core and the impact of barrier latency decreases. With 16 cores (left half of Figure 22), the difference in performance between TLSync and other barrier implementations almost disappears when the input size reaches 32,768. However, when the number of cores is increased to 256 (right half of Figure 22), the same work is split among more cores and the impact of barrier latency increases—even with the input size of 65,536, the TLSync barrier provides speedups of at least 2X relative to tournament and Barrier Filter barriers. I also observe that barrier latency has a larger impact with large cores, because they finish the work between barriers faster and experience longer barrier latencies. This means that, as technology scales, barrier latency will have more impact regardless of whether the increased transistor count is used to improve performance of each individual core, to put more cores on



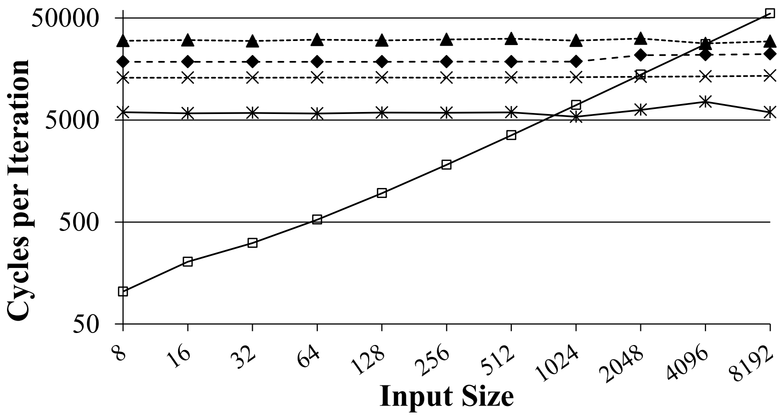
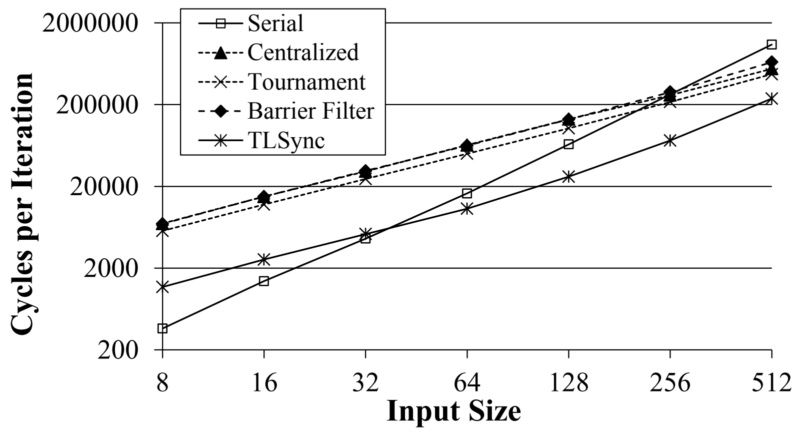


Figure 23: Performance of Livermore Loop 3 with various barrier mechanisms with 16 (top) and 256 (bottom) cores.

the chip, or for a combination of both.

Figure 23 shows the results of experiments on Livermore kernel 3. Small and large cores have nearly identical trends, so I omitted Small Core results for brevity.

With 16 cores, the trends are similar to those observed for Livermore Loop 2. Loop 3 computes a simple inner product of two vectors and is parallelized by computing a partial sum-of-products in each thread, then combining the partial sums in the "main" thread. With 256 cores, the parallel portion of the execution is finished much faster than with 16 threads, but the serial part takes longer (256 partial sums to add instead of 16) and dominates the execution time until input sizes become sufficiently large. This causes "Serial" execution to perform better than any of the parallel executions until input sizes reach about 512. The same effect reduces the



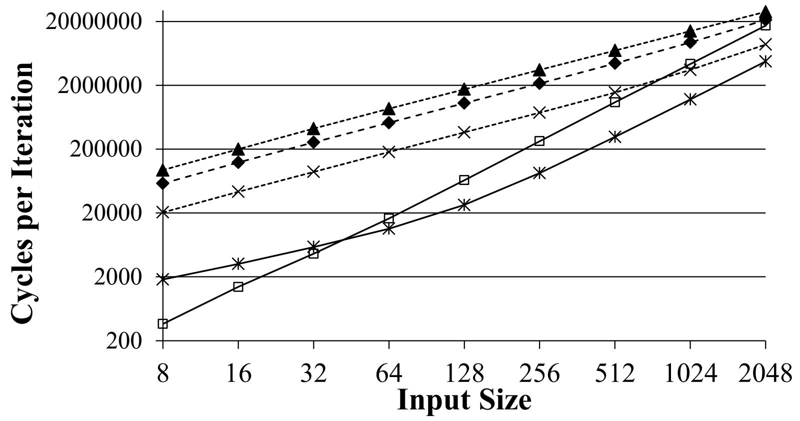


Figure 24: Performance of Livermore Loop 6 with various barrier mechanisms with 16 (top) and 256 (bottom) cores.

impact of barrier latency for small inputs and moves the convergence point for their performance beyond the 8192 input size.

Livermore Loop 6 is a reduction for a general linear equation, and is parallelized using the coordinate axis transformation [80]. The results for Loop 6 are shown in Figure 24. Again, I only show Large Core results for brevity because Small core results show nearly identical trends.

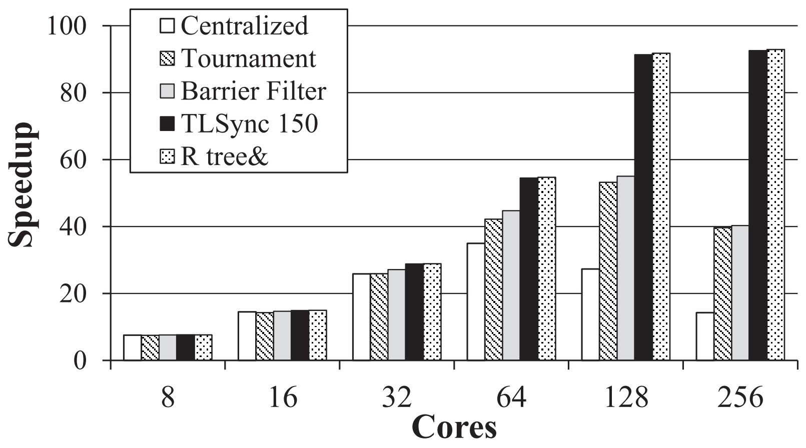
The trends for 16-core execution are nearly identical to those for Loop 2 and Loop3—the lower barrier latency in TLSync results in better overall performance, with a trend toward eventual convergence as input sizes increase. When the core count increases to 256, the difference between TLSync and longer-latency barrier implementations increases because each core has less work between barriers and the barrier latency becomes more significant.

In addition to lowering execution times, an important effect of a low-latency barrier implementation is that it reduces the input size necessary to achieve a parallel speedup. In all three Livermore loops (Figures 22, 23 and 24), I observe that with TL-Sync barriers the break-even point (where the parallel execution time is equal to serial execution time) occurs at smaller input sizes than with longer-latency barrier implementations. This is very important for parallelization of non-scientific applications—unlike scientific applications where ever-larger input sizes are the norm, input sizes for many consumer applications are either determined by standards (e.g., for video and audio) or grow less quickly than core counts due to other limitations, e.g., display resolution and human perception thresholds for games.

5.3.3 Streamcluster and Dithering

To examine the impact of various barrier mechanisms on overall performance in non-scientific applications, I use the Streamcluster benchmark from PARSEC [15] and the Dithering benchmark from the EEMBC [29] benchmark suite. Streamcluster is an RMS kernel that solves an online clustering problem using a streaming algorithm. To synchronize threads between relatively short tasks along the stream, it uses a large number of barriers (8,772 for 1K data points). Dithering is an embedded benchmark that converts an image into a error-diffused image suitable for printing using the Floyd-Steinberg dithering algorithm. I hand-parallelized the application to exploit fine-grained parallelism by processing each row concurrently. In this implementation, a pair of barriers were used to synchronize error propagation between rows.

Figure 25 shows the parallel speedup of the two applications for different barrier implementations and numbers of cores. I used 1K data points for the Streamcluster, and a 4,096×3,072 image for the Dithering application. Each application executed



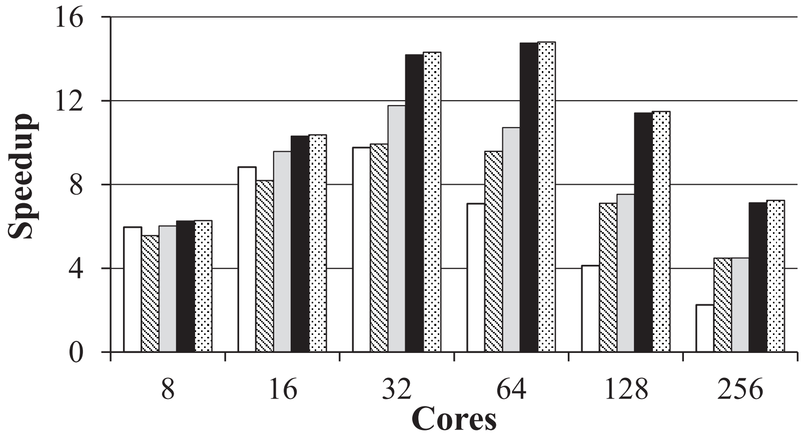


Figure 25: Performance of Dithering (top) and Streamcluster (bottom) with various barrier mechanisms.

8,772 and 6,144, respectively. In addition to the centralized, tournament, and Barrier Filter barrier implementations, I compare TLSync performance to a hardware barrier with reduction and notification trees implemented with traditional wires with optimally-spaced repeaters (shown as "R Tree&") Recall that this hardware barrier implementation requires a separate barrier tree for each barrier group, and that the optimally-repeated chip-spanning wiring of this implementation requires repeater placement at frequent intervals along each wire, which interferes with the layout of "local" logic (e.g., cores and caches) that the wire is traversing. I observe that hardware barriers (both variants of TLSync and "R Tree&") show nearly identical performance than software-only and Barrier Filter implementations. This result is in-line with the results from Livermore loops, and leads to two important conclusions

Table 16: The cost of main active components used in TLSync for a 16-core processor in 45nm technology.

Active Element	Area (mm^2)	Power (mW)	# Used	Total Area (mm^2)	Total Power (W)
Amplifier [16]	0.008	8	9	0.08	0.07
Mixer [85]	0.005	1.46	16	0.08	0.02
Filter [91, 102]	0.07	10	16	1.12	0.16
AM demodulator [96]	0.002	3.03	16	0.02	0.05
Total	–	–	–	1.30	0.30

regarding the performance impact of barriers. First, various scalable low-latency barriers have similar performance on real workloads in spite of differences in actual barrier latency—when barrier latency is low enough (tens of cycles or less), execution time between barriers dominates the execution time and further reduction in barrier latency provides minimal returns. Second, the advantage of scalable low-latency barriers increases significantly as the core count increases, not only up to the point when the application performance stops scaling for other reasons (e.g., Streamcluster), but also beyond the point (e.g., Dithering). As noted earlier, this is because larger core counts result in less work per core between barriers, turning barrier latencies into a larger fraction of overall execution time, and thus increasing the performance impact of barriers.

5.3.4 Implementation Cost

Transmission line (TL) broadcast network with TLSync barrier support has two main sources of cost—the use of metal layers for the actual TL, and the use of silicon area for active components of the design.

In terms of metal area, a TL should be implemented in the top (global) metal layers. Because a TL is shielded from the circuitry below by its "ground plane" conductor, it can easily be routed over any local circuitry. Additionally, TL broadcast network only requires amplifiers at connection points between segments, at millimeter-scale distances from each other. This allows the TL network to be routed without affecting the placement of circuitry within each core.

The main drawback of using TLs is in their large width: a stripline implementation occupies a 12μm-wide strip in three metal layers. Although the main conductor is only 6μm wide, an additional 3μm on each side must be free of other wires or TLs to keep crosstalk within tolerable bounds. This metal use is significantly larger than for traditional global wires, which have only a 0.135μm pitch in 45nm technology, with proportional scaling in future technologies. As explained in Section 5.1, a single chip-spanning TL broadcast tree can support multiple barrier groups with a very low latency, and the total metal area needed for this tree in 45nm technology is only 1.7mm^2—two lines (input and output) for each segment, 24mm total length of all segments, using 12μm each in three layers for a total area of 2×24mm×0.012mm×3. Although this appears expensive, it should be noted that it represents only 0.07% of the total metal area for a 16mm×16mm chip with 10 metal layers. Furthermore, the only requirement for "ground plane" conductors is that they should not carry RF signals, so these wide conductors can be leveraged for Vdd and Vss distribution. Alternatively, if the metal layers used for Vdd and Vss are placed above and below the TL's main conductor metal layer, there would be no need for separate "ground plane" conductors, thus reducing the metal area demands for TLs by a factor of three.

The main active components in TLSync are the amplifiers between transmission line segments, and the mixer, filter, and AM demodulator (amplitude tracker) circuitry in each receiver. Table 16 summarizes the chip area occupied by each active component, together with energy consumed by that component. I use the resistive-feedback low-noise amplifier with a high gain from [16], the low-voltage ultra-wideband down-conversion mixer from [85], the ultra-compact active bandpass filter from [102] with low-power enhancements from [91], and the compact CMOS-based amplitude tracker from [96]. Because the original filter and amplitude tracker were built in 180nm and 350nm CMOS technology, their area and energy numbers were scaled to 45nm technology using conservative assumptions: I assumed that area is reduced in linear (not quadratic) proportion to feature size, and that power is reduced only because of the reduction in supply voltage (this preserves the current in analog circuitry to ensure its correct operation after scaling). Note that all components are chosen to scale well and avoid the use of large passive (e.g., inductor, capacitor, and resistor) components, so their actual scaling is likely to be better than described above.

CHAPTER VI

HYBRID INTERCONNECT: LOW-LATENCY UNSWITCHED TL RING AND A HIGH-THROUGHPUT SWITCHED INTERCONNECT

6.1 Motivation

The steady increase in on-chip core count has already reached the point where a shared bus no longer provides sufficient bandwidth for the coherence traffic. Further, bus latency and energy-per-bit are not scaling favorably. Consequently, a many-core chip must use a packet-switched network-on-chip (NoCs), e.g. a mesh or torus [26]. In such NoCs, each link is short and its length scales down with technology, so latency and energy consumption scale well if most traffic is local. Further, the aggregate bandwidth of all the links scales with the number of cores. However, switched networks come at a significant latency cost for traffic between far-apart cores—these messages traverse more switches when the core count grows, in addition to longer wire delays due to technology scaling.

One approach for reducing NoC latency is to use transmission lines (TLs), where signals propagate close to the speed of light. However, TLs consume far more metal area than traditional wires, so a same-area TL-based NoC typically either provides far less throughput or requires very sophisticated signaling to improve this throughput.

The key insight that guides my approach in this chapter is that the latency advantage of TLs over wires is the greatest for long-distance packets, while local traffic (which often represents the majority of the traffic) sees a relatively low latency even in a traditional mesh based NoC and it has little benefit from going over (expensive) TLs. For example, as shown in Figure 26, applications tend to show heavy local traffic

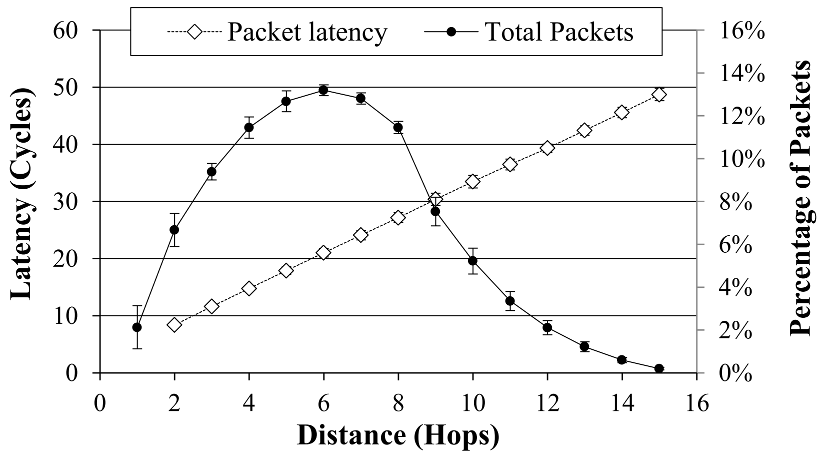


Figure 26: Distribution of the number of packets and packet latency on switched interconnection for varying distances. Values are averaged for all the tested applications and shown with 95% confidence interval.

(42% of total packets travel less than 5 hops) where the packet latencies are less than 20 cycles. Meanwhile, only 7.5% of the packets suffer from high packet latencies while passing through more than 10 hops. Guided by this insight, I design a low-cost unidirectional TL (UTL) ring interconnect that provides very low maximum propagation latency (e.g. 2 ns for a 64-core chip), uses simple signaling and has a very efficient and simple arbitration mechanism. However, this ring has limited throughput, so I use it *together* with a traditional switched NoC, by judiciously steering each packet to the ring or the packet-switched interconnect.

6.2 Transmission Line Ring

This section presents a detailed design of a TL-based low-latency ring. Because TLs are expensive and have extremely fast propagation, I chose a topology that minimizes overall TL length rather than the longest path—a ring (Figure 27) rather than a double tree [73] or a grid [1]. The models in Section 6.2.1 show that the entire ring can still be traversed in 1.6 ns (64-core chip in 22 nm technology) to 2.6 ns (256-core chip, 10 nm technology). The latency gain from a lower-latency topology would be only 1–2 cycles, but the cost would be several times higher. Furthermore, a ring

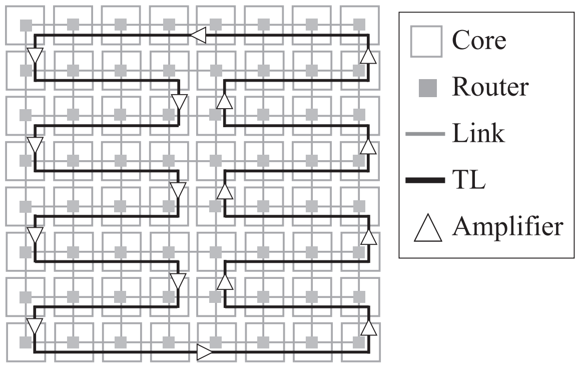


Figure 27: TL-based ring with packet-switched network.

design allows a fast arbitration approach (Section 6.2.2).

6.2.1 Unidirectional Transmission Line Ring

The TL ring is implemented by chaining unidirectional transmission line (UTL) elements and periodically using amplifiers to make up for losses along the ring. Following the discussion of the UTL elements and how they are connected into a ring (Section 6.2.1.1), I discuss how to prevent the signal from infinitely looping around the ring (Section 6.2.1.2) and how data bits are transmitted and received on this ring (Section 6.2.1.3).

6.2.1.1 UTL-Based Ring Design.

The goal of the TL design here is to manage two main problems that traditional TL designs with many connections (for transmitters and receivers) [73] face—attenuation of signal power due to split at each connection and signal reflections/reverberations due to connections and other discontinuities (e.g. slight impedance mismatches along the TL). With naïve connections like T-junctions, significant reflection cannot be avoided. Worse, the optimal design from reflection standpoint is to split the remaining signal power half-half, but it weakens the forward-propagating signal quickly as the signal goes through multiple connections. Soon, the "noise" caused by the sum of the many reflections of reverberating signal (with various delays and attenuations) between the connections will dominate the weakened signal. As a result, the number of connections that are feasible for one TL segment is low, especially for high frequencies: 6–8 connections (two at the ends of the TL and 4–6 along the way) up to 10 GHz frequency and even fewer at higher frequencies (but they allow high data rates).

Because of these considerations, I design a new TL structure that provides uni-directional propagation. This way, reflections cannot reverberate through the TL medium and more signal power can continue to propagate along the ring at each connection. The new structure is based on a four-port coupler, which is traditionally used in microwave designs to reduce reflection issues in signal split. The coupler used in this study is based on a recent design [28] that has excellent directivity and frequency characteristics. A simple coupler consists of two TL structures running in parallel close to each other, as shown in Figure 28a. If the signal is injected into one end of one of the lines (e.g. ① in Figure 28a), it propagates to the other end, but some part of the signal's power is transferred to the other line through EM coupling. Interestingly, with careful design, it is possible to achieve excellent *directivity* for such a coupler. In this example, almost all of the transferred power goes to port ③ on the bottom line while nearly nothing goes to port ④. Note that the coupler is symmetric and any of the four connections can be used as "input". For example, if the signal is injected into the port ④ instead of the port ① in Figure 28a, it still splits between the right ends of the two lines (port ② and ③).

This allows us to chain couplers together using one of the lines as the "main" TL and the other line to connect transmitters and receivers (Figure 28b). Traditional couplers are designed to split the signal equally between the outputs and the signal on the "main" line loses half of its power at each coupler. This would require amplifiers to be placed frequently along the ring. I modify the coupler design so that most of the signal's power continues down to the "main" TL and only 25% of the power goes to the receiver. This allows us to save half of the amplifiers.

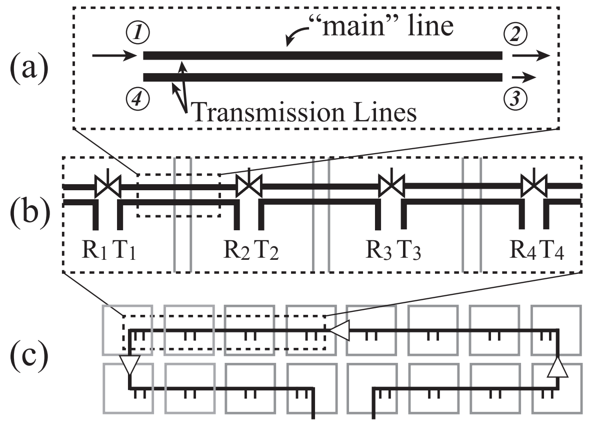


Figure 28: Connecting UTL couplers into a ring.

To achieve improved frequency, directionality and power transfer properties, in this coupler, each of the two "lines" consists of two conductors that are connected at their endpoints as shown in Figure 29. I still favor this design over a regular TL because it simplifies implementation and reduces the circuitry needed in each node and along the ring.

6.2.1.2 Infinite Loop Prevention.

Amplifiers in the ring compensate for losses in signal strength. So the signal, once injected by a transmitter, would continue to circle around the ring infinitely and prohibit transmission of new signals. To prevent such infinite loops, each transmitter controls a pass-gate that connects the previous coupler's "main" ring connection to its own. This pass-gate is normally in the connected state, i.e., the signal is allowed to propagate freely. However, when a transmitter is about to transmit a packet, it first puts the pass-gate it controls into the disconnected state. This transforms the ring into a chain so the signal reaches all the other nodes but does not continue along the ring for the second time. Note that each pass-gate only introduces a single transistor delay into the signal's path. This delay improves with technology scaling, making these pass-gates both cost- and energy-efficient.

As shown in Figure 28b, the receiver of each node is connected before the node's

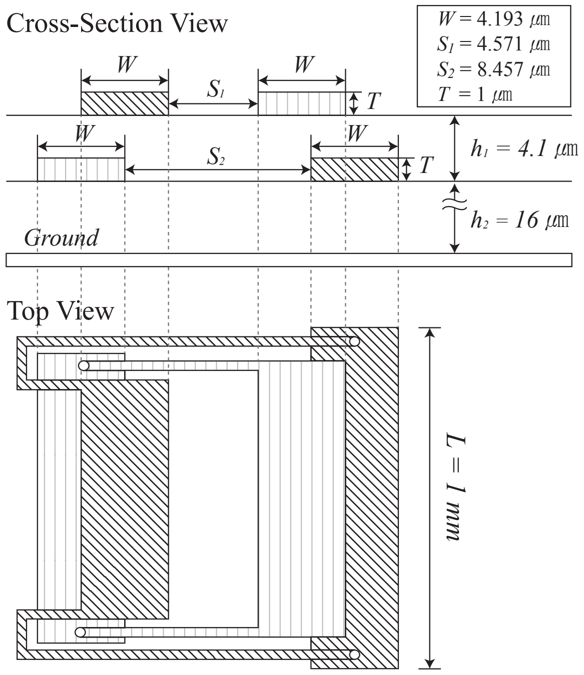


Figure 29: Cross-section and top view of the coupler.

pass gate, so sender is the last to receives its own signal. This can be used for error detection, although I omit that discussion because the ring has low error rates (10^{-15} or less). More importantly, this arrangement allows a new arbitration approach (Section 6.2.2).

6.2.1.3 Signal (De)Modulation.

The modulation method is a tradeoff between receiver/transmitter complexity, their latency, error rates, and the bit rates that can be achieved within a given frequency band. Typically, high bit rates require modulations with multiple bits per "symbol" (e.g. signal level in AM modulation) with more complex and longer-delay transmitters/receivers. Also, such modulations have smaller differences between symbols. Furthermore, they tend to require complex and long-delay approaches to reduce the error rates.

The primary goal for the TL ring is to minimize its latency and cost. Consequently,

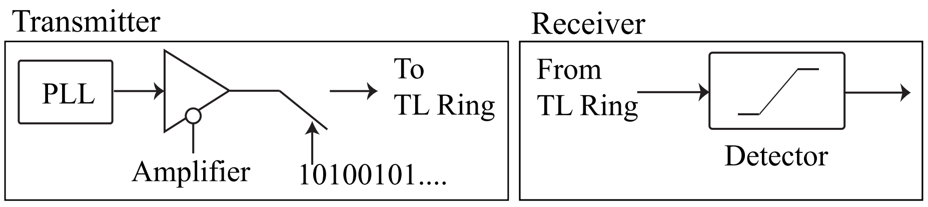


Figure 30: Transmitter and receiver design for TL ring.

I use a very simple form of ultra-wideband modulation: on-off keying of a high-frequency clock signal. With this modulation, one or more consecutive pulses of the clock represent one bit, and these pulses are either transmitted (bit is one) or not (bit is zero).

The pulse frequency and bit rate should correspond to the central frequency and half-width of the available spectrum of the transmission medium. The coupler design in this study has available spectrum that scales up as its length scales down. For 1 mm length, the available spectrum is 20–60 GHz, so I use a 40 GHz signal clock with two pulses per bit (up to 20 Gbits/s). As cores get smaller (and transistors faster) with technology scaling, this design can be modified to use 0.5 mm coupler length. The available spectrum would be 40–120 GHz, so 40 Gbits/s can be transmitted using an 80 GHz signal clock and two pulses per bit.

The receiver and transmitter for this modulation approach are simple (Figure 30). The transmitter contains (1) a PLL to generate pulses, (2) a pass-gate to control whether the pulses get onto the transmission line and (3) an amplifier (composed of two appropriately sized CMOS inverters) that mostly provides impedance matching. The receiver consists of two components: a detector (capacitor and a few CMOS transistors) to identify if a pulse is present in each bit position and a fast shift register (clocked by the same PLL used for the transmitter) to collect the high-rate bits so they can be used by logic operating at “normal” on-chip clock frequencies (e.g. 1GHz). The PLLs of all transmitters can be phase-locked to the existing skew-compensated global clock signal, which is used for mesh routers and cores to derive the clock signal.

6.2.1.4 End-to-End Latency on the Ring.

The latency has five components: 1) TL propagation time that largely depends on the total distance traveled between the farthest-apart transmitter and receiver, 2) transistor circuit latencies that depend on the complexity of the circuitry in transmitters, amplifiers, and receivers, which altogether continues to improve with technology scaling, 3) delays introduced by fundamental requirements of signal processing in detectors, and 4) packet transmission latency that depends on the number of bits in the packet and the time needed to transmit one bit (Section 6.2.1.3), and finally 5) queuing and arbitration delays that I addressed in Sections 6.3.2 and 6.2.2 respectively.

Now, I summarize the first three components of the latency: first for the transmitter, then for the TL and amplifiers on the ring, and finally for the receiver.

Transmitter delay consists of a single CMOS pass-gate and an amplifier (Figure 30) that connect the PLL to the transmit port of the TL coupler. The pass-gate delay is only a few picoseconds and improves with technology scaling. The amplifier latency is about 50 ps in 45 nm technology and scales with technology [88] (I conservatively skip the 32 nm technology node and assume 25 ps for 22 nm and 13 ps for 10 nm technology). Note that the amplifier is needed mainly for impedance matching if the PLL signal is strong enough. In the simulations, I find that this amplifier can be replaced by an inverter, with transistors sized for 50 Ω output impedance.

Propagation speed along the UTL is 0.0075 ns/mm (from the AWR [6] EM simulator). For a 16 mm$\times$16 mm die with 64 cores, the ring is 156.4 mm long, with 16 amplifiers along the way. When the number of cores is increased to 256 (in 10 nm technology), the ring length is 286.4 mm with 32 (faster) amplifiers along the way. Hence, the propagation delay (transmitter output to receiver input) is expected to be 1.6 ns for a 64-core chip in 22 nm technology (156.4 mm$\cdot$0.0075 ns/mm+16 amps$\cdot$0.025 ns/amp), and 2.6 ns for a 256-core chip in 10 nm (286.4 mm $\cdot$0.0075 ns/mm+32 amps$\cdot$0.013 ns/amp).

At the receiver, the delay is dominated by the delay of a detector, which is a signal-processing delay determined by the frequency of the received RF signal. For the 40 GHz AM detector used in this design, the delay was 25 ps (one pulse).

6.2.2 Disconnect-Based Arbitration Mechanism

With a separate arbitrator, the request/grant signals must be sent to/from the arbitrator by a separate set of TLs (which is expensive), or through long-latency wires (which leads to much longer overall latency for ring-steered packets). Instead, I use the ring itself for arbitration by adapting the Token Ring approach from nanophotonic broadcast rings [97]—the sender transmits a token down the ring when it is done, and the first prospective sender node that gets the token becomes the new sender. However, in nanophotonic interconnects, a dedicated wavelength is used for the token signal, whereas TL ring with simple modulation uses all of its bandwidth for one channel. Therefore, I attach a token sequence at the end of the packet. The token sequence consists of a few disconnect-delay bits, a token bit that is initially equal to 1, and a few reconnect-delay bits. Each potential sender looks for the end of the current packet, disconnects the ring using its pass-gate and receives the token bit. If the token bit is zero, it reconnects the ring. This way, only the first potential sender will receive the token bit with a value of 1—the disconnect in this modulation scheme results in a 0 value (no pulses) continuing down the ring so other prospective senders receive 0 value for the arbitration bit. EM and circuit simulations show that disconnect and reconnect delays of only 2 bits are sufficient, so arbitration “spends” only five bits worth of throughput per packet.

6.2.3 Broadcasts on the TL Ring

A packet sent on the TL ring is “seen” by all other cores, so I convert multicast packets (e.g., when sending out invalidations) into broadcasts. However, the baseline directory protocol is used for all coherence actions, even though a ring can be used as an ordered interconnect to support efficient snooping [65]. In my approach, most packets still use an unordered (mesh) interconnect, which would lead to ordering inconsistencies if both snoopy and the directory-based actions can occur on the same cache block. A solution for this is outside the scope of this thesis and I leave it as future work.

6.3 Traffic Steering

The UTL ring presented in Section 6.2 provides low-latency and efficient arbitration, but its aggregate throughput is significantly lower than that of wire-based switched NoCs. Instead of boosting the ring throughput using multiple rings, sophisticated modulation and even higher signaling frequencies that cause major increase in cost and design complexity, I combine the low-latency-but-limited-throughput UTL ring with a traditional switched interconnect. This approach is based on an insight that the latency benefit from using the ring varies from message to message, so one can boost the benefit from the low-latency ring by using it only for packets that are expected to show a lot of benefit from the reduced latency while the remaining (majority of) packets continue to use the high-throughput switched interconnect.

My steering approach has three components: (1) a scoring mechanism that predicts the benefit from ring-steering a given packet (Section 6.3.1), (2) a threshold management mechanism that controls ring throughput consumption (Section 6.3.2) and (3) an emergency re-steering mechanism for sudden rises in ring contention (Section 6.3.3).

6.3.1 Benefit Estimation and Packet Scoring

This scheme rates each packet and assigns a score according to how much benefit it can expect from being steered to the ring instead of the mesh. This score S is a sum of two components: latency benefit S_{lat} and criticality score S_{crit}. The latency benefit S_{lat} is the difference in packet latency, which requires future knowledge and is thus estimated. So, $S_{lat} \approx L_{mesh} - L_{ring}$, where L_{mesh} and L_{ring} are the estimated latencies for the mesh and the ring, respectively.

Because the main latency factors in the ring and mesh are different, I have a separate estimator for each. For the mesh interconnect, the latency depends on the packet's hop count and on congestion in the network. Without congestion, latency is proportional to hop count. However, in my experiments, a 5-hop packet exhibits a latency from about 19 cycles (without contention) to 56 cycles (with heavy contention), so it is important to account for contention when estimating L_{mesh}. For the TL ring, the range of packet latencies can be even larger without some form of demand control. Ring-steered packet latency can be as low as 4–6 cycles without contention (4 cycles to transmit a 64-bit packet at 16 Gb/s, plus 0–2 cycles of propagation latency depending on destination). In my experiments with *lu-cn*, most packets indeed have reasonable latency (82.7% are below 20 cycles) but some packets (about 5%) have latencies above 100 cycles (and up to 2600 cycles) due to contention when I randomly steer packets to TL ring with 50% chance.

The mesh latency estimator is based on caching actual latencies for recent packets that had the same hop count. This is implemented with small cache (*m-cache*) in each node. The *m-cache* has s rows with one row for each possible hop count in the mesh. Each row stores n most recently observed latencies for that hop count for messages sent by that node. In my experiments, $s = 16$ (8×8 mesh for 64-cores) and $n = 4$. I use 8-bit *m-cache* latency values (using 255 for latencies that are too large to represent with 8 bits), resulting in only 64 bytes of *m-cache* state per core. I use three latency prediction strategies: (1) most recent latency, (2) average of latest $\frac{n}{2}$ latencies and (3) average of all n latencies. All three use the same state (the *m-cache*), and they offer a tradeoff between quick adaptation and ability to filter outliers. Each node picks the best-performing strategy as L_{mesh} for its score calculation, and tracks accuracy for each strategy with a saturating counter that is incremented by 2 when that strategy's prediction is the closest to actual latency and decremented by 1 otherwise. To do this, I add short timestamps to outgoing packets, compute actual latency at the destination node, which then piggybacks this information to a packet that travels in the opposite direction.

The ring latency estimator exploits the broadcast nature of the ring to estimate contention without sending any additional bits on the ring. The latency estimator considers the contention-free latency to the receiver as well as two contention-related factors: 1) probability of finding the ring idle (p_{idle}) and 2) the expected time I have to wait if the ring is not idle (t_{queue}). After I estimate p_{idle} and t_{queue}, I can estimate the packet's overall latency L_{ring} on the ring as $L_{ring} = l + t_{queue} \times (1 - p_{free})$, where l is the expected time to transmit a single packet (and its arbitration sequence) on a contention-free ring.

To estimate p_{idle} and t_{queue}, for the most recent K packets seen on the ring, each node records 1) the number of elapsed cycles since the previous packet, i.e. $t_k = T_k - T_{k-1}$ and 2) the distance (d_k) along the ring between the previous and the current transmitter. The probability of finding the ring idle is estimated using the history of elapsed time: over the last K ring accesses the total elapsed time was $\sum_{i=1}^{K} t_k$ cycles while the ring was busy for $l \times K$ cycles. Therefore, the ring is expected to be free with a probability of $p_{free} = \left(1 - \frac{l \times K}{\sum_{i=1}^{K} t_i}\right)$.

To estimate t_{queue}, I first estimate p_{core}, the probability that a core will use the ring if given the opportunity ("live" token with arbitration bit **1** passes by). Over the last K packets, the number of cores that were given this opportunity was $\sum_{i=1}^{K} d_i$, but only K of them used the ring. Thus I estimate $p_{core} = \frac{K}{\sum_{i=1}^{K} d_i}$. If the ring-distance between the current transmitter and this node is k, the expected number of packets that will be transmitted before this node can get the "live" token is $k \times p_{core}$, and the expected wait time for a packet on this node is $l \times k \times p_{core}$. Note that this assumes no packets are already queued for the ring at this node. If there are w already-queued packets on this node, the first of those packets will be transmitted when this node gets the live token, and then for each subsequent packet the node will have to wait until every node around the ring gets the opportunity to transmit—if there are N cores in the ring, each "circle" around the ring is expected to take l cycles for our own packet (which I know will be sent) plus $l \times (N-1) \times p_{core}$ cycles for packets from other $N-1$ nodes. The overall waiting time for a packet is then $t_{queue} = l \times k \times p_{core} + w \times (l + l \times (N-1) \times p_{core})$.

The criticality adjustment S_{crit} is used to account for known and/or estimated effects of the packet's latency on the actual performance. In principle, one can use instruction-level criticality estimator that attributes some of the message's latency to overall stall time and uses it for future messages from the same instruction. To simplify hardware, I use a simple approach that assigns a constant penalty to messages that are unlikely to be critical. Examples include write-back messages, speculative replies and their confirmations when various protocol optimizations are used.

6.3.2 Threshold-Based Throughput Control

Note that the overall latency benefit of having the TL ring can be roughly estimated as the product of the number of ring-steered messages and the average latency benefit per message. Without any throughput control, too many packets would be steered to the ring under high-load conditions. An equilibrium would eventually be reached at a point where ring contention is so high that it offsets the latency advantage for most messages. The result would be that the ring still carries a small fraction of the traffic while provides very little latency advantage for that traffic. Thus, my throughput mechanism aims to maintain a ring utilization level that is low enough to preserve its low latency, but high enough not to waste the ring throughput. The "Goldilocks" utilization level depends on the distribution of mesh latencies in the application, efficiency of arbitration on the ring and many other factors. However, I found that target utilizations around 75% tend to work well for the TL ring—it keeps queuing delay at a low level, but still utilizes a significant portion of the available throughput of the ring.

My approach to controlling utilization of the ring is to maintain a threshold for scores used in steering decisions. Instead of steering packets based on whether its expected-benefit score is positive or negative, I now steer a packet only when its score exceeds the threshold. To control ring utilization, I adjust the threshold upward when the ring is busier than desired, or decrease the threshold if the ring utilization is too low.

The ring utilization is measured locally at each core using a TL ring access counter. Each core increments the counter when a packet is observed. For every T cycles, the counter is destructively read and ring utilization is calculated as $\frac{l \times n}{T}$, where the counter value is n and the TL ring propagation latency is l. In my implementation, I update utilization every 512 cycles ($T = 512$), and increment or decrement the threshold by 1 if the utilization is above or below a preset target utilization.

Finally, note that the high ring utilization also creates significant priority inversion problems—a high-scored message on one core waits for other cores' lower-scored messages that are sent on the ring already. This priority-inversion problem can get resolved with score-aware arbitration at a cost of more ring throughput wasted on arbitration. However, when ring utilization is lower (e.g. below 85%) such priority inversion is rare, so score-aware arbitration is counter-productive—the penalty from larger arbitration sequences is higher than the benefit from eliminating priority inversion.

6.3.3 Emergency Re-steering

This steering mechanism is based on estimated latencies. If the actual latency of TL-steered packet is higher than estimated, e.g. due to a sudden burst of ring-steered messages, it may take a while for the throughput control to compensate. During that time, ring-steered messages may experience very long latencies. To avoid this, I implement an emergency re-steering mechanism, which periodically checks entries in the local TL packet queue and resteers a packet to the mesh if it has waited too long. To find such packets without keeping much state, I use a single bit in each TL packet queue entry. This bit is set to 0 when the packet is enqueued. Periodically, I check these bits. If the bit is 0, it is set to 1. If the bit is already set, the packet is resteered. With this scheme, if checks are performed every T cycles, the maximum penalty from ring-steering a packet is limited to $2 \times T$.

6.4 Implementation Cost

Transmission-line (TL) ring has two main sources of cost—the use of metal layers for the coupler-based TL ring, and the use of silicon area for active components of the design.

In terms of metal area, a coupler-based TL should be implemented in the top two (global) metal layers. However, the "ground plane" conductor should be in the lowest global metal layer, and the metal layers in between these two should be clear of metal and cannot be used for any local circuitry. Hence, all global metal layers need to be used for realization of coupler-based TLs. In comparison, TL ring only requires switches at connection points between couplers, while amplifiers are required at connection points after every eight couplers. This sporadically used switches and amplifiers with millimeter-scale distances between them allows the TL ring to be placed-and-routed without affecting the layout of circuitry within each core.

The main drawback of using transmission lines is in their large width—a microstrip coupler implementation occupies a 20 μm-wide area in two metal layers. The coupler consists of two strips, each only 4 μm wide, but to achieve the desired amount of coupling, the spacing between the strips has to be about 10 μm. Furthermore, other wires or transmission lines cannot go between its "ground plane" conductors without

Table 17: The cost of main active components used in TL ring for 64-core processor in 22 nm technology.

Active Element	**Area (mm^2)**	**Power (mW)**	**# Used**	**Total Area (mm^2)**	**Total Power (W)**
Amplifier [16]	0.017	28	16	0.272	0.448
Detector [74]	0.00024	0.84	64	0.015	0.054
Total	–	–	–	0.287	0.502

creating significant crosstalk. This metal use is significantly larger than traditional global wires that have only a 0.135 μm pitch in 45 nm technology with proportional scaling in future technologies. This difference in metal area consumption reinforces the argument in favor of *combining* a fast TL based ring with the traditional switched interconnects, rather than trying to replace a traditional high-bandwidth interconnect with a much more expensive TL-based interconnect.

A conservative estimate of the total metal area necessary for the TL ring with 64 processors in 22 nm technology is 31.28 mm^2—two coupled lines (transmitter, receiver and the next coupler) for each coupler, 156.4 mm total length of the TL ring, 20 μm total width in 10 layers for a total of 156.4 mm$\times$0.020 mm$\times$10. Although this appears expensive, it should be noted that it represents only 1.22% of the total metal area for a 16 mm$\times$16 mm chip with 10 metal layers. Furthermore, the only requirement for "ground plane" conductor is that it should not carry RF signals, so this wide conductor can be leveraged for Vdd or Vss distribution.

The main active components are the amplifiers between transmission line segments and amplitude detector in each receiver (Table 17). For amplifier design, I model a low-noise amplifier (LNA) that can be implemented in a CMOS process [16]. In the original 45 nm implementation, this amplifier occupies 0.017 mm^2 of chip area. For the detector, I model a compact CMOS-based design [74]. In the original 90 nm technology, this detector only occupies 0.00024 mm^2 of chip area, and can be expected to scale well because it entirely consists of transistors and very small capacitors and inductors.

6.5 Performance Evaluation

For performance evaluation, I implement a detailed model of the TL ring's message and arbitration timing as well as its traffic steering in SESC [79], an open-source cycle-accurate architectural simulator. The baseline system is a 64-tile CMP system. A tile contains a 2-issue out-of-order core at 1 GHz, a 32KB instruction L1 cache, a 32KB L1 data cache, a 1MB slice of the L2 cache, a directory slice and a mesh router. The latency of an off-chip memory access is 200 cycle, with memory controllers located in each corner of the chip. The packet-switched interconnects are modeled in detail using the cycle accurate Booksim simulator [25], which I integrated into SESC. The evaluation is based on 8×8 2D mesh with 128 bit channel with one cycle delay. I model 8 virtual channels (24 buffers) per router port and packets are routed using Dimension-Order Routing (DOR).

6.5.1 Characteristics of the Benchmark Applications

I evaluate my scheme with parallel applications from SPLASH-2 [99] and PARSEC 3 [13] benchmark suites. For SPLASH-2, enlarged input set sizes are used to achieve execution of at least 1 billion instructions. For PARSEC, I use sim-medium inputs. The execution time is measured only in the parallel section as intended by the benchmark designers, denoted as region of interest.

6.5.1.1 Latency-critical Applications

Out of 20 applications in SPLASH-2 and PARSEC 3, not all benefit from better on-chip interconnects. Figure 31 shows that even with ideal interconnect that can transmit any packets in a single cycle with infinite throughput, half of the applications show less than 20% improvement in overall application performance. These applications either suffer from high rates of off-chip memory access (e.g., *fft* and *radix* show 98.7% and 85.0% of accesses go off-chip), or exploit memory- or instruction-level

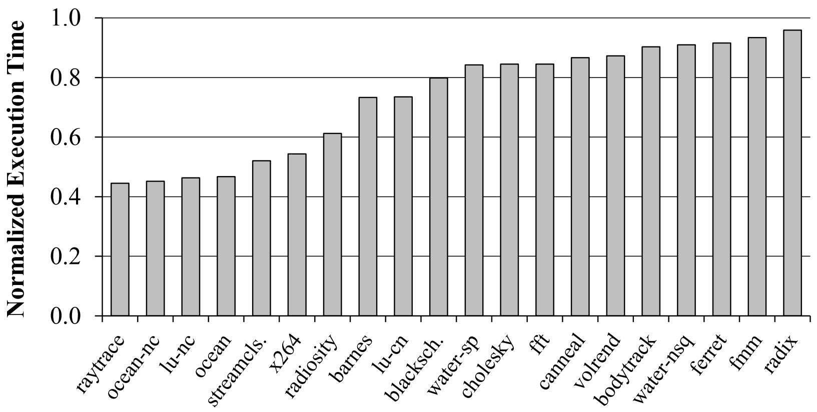
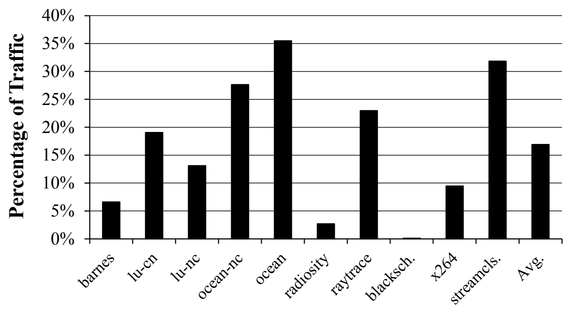


Figure 31: Normalized execution time with infinite throughput on-chip interconnect (single cycle latency).

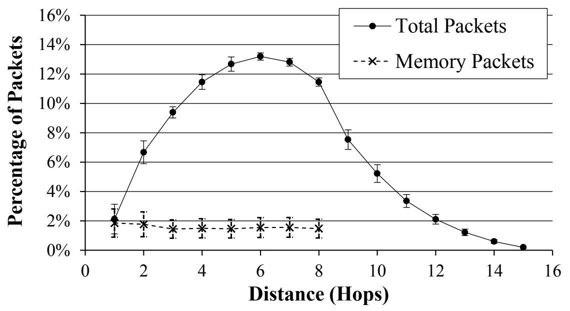
parallelism to hide on-chip access latencies. In the remainder of this evaluation, I focus on applications whose performance is sensitive to NoC characteristics. Thus, only applications that have at least a 20% execution time reduction with ideal interconnect is used (called *latency-critical applications*).

6.5.1.2 Memory Traffic Pattern

Memory access traffic consists of memory request packets destined to on-chip memory controllers and reply packets with requested cache lines. The memory traffic takes 16.9% of the total on-chip traffic on average, and increases up to 35.5% for *ocean* as shown in Figure 32a. The effect of this memory traffic on application performance depends on how long the packets travel that is largely determined by the geographical location of the on-chip memory controllers. In this study, 8 memory controllers are placed on top of the first row in the 8×8 mesh grid. The home nodes and memory addresses are carefully mapped to limit the distance between a requesting home node and corresponding memory controller to 8 hops as shown in Figure 32b. Figure 33 depicts the significantly higher memory traffic at core 1 to 8 due to the memory controllers. Despite of the considerable amount of memory traffic, the steering algorithm directs almost all of the memory packets carrying cache lines to a switched network because those packets show no benefit with the TL ring. Here, the limited bandwidth

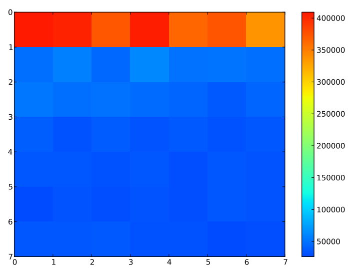


(a) Percentage of memory access traffic

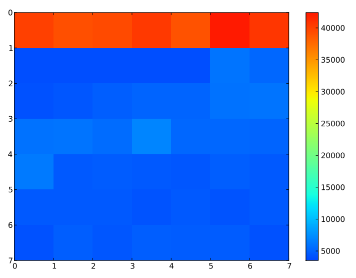


(b) Distribution of memory access packets with 95% confidence interval

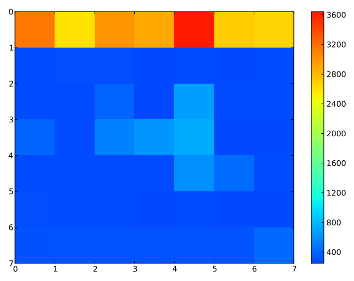
Figure 32: Total packets and memory access packets on switched on-chip interconnect with 95% confidence interval. 8 memory controllers are located above the top row of the mesh network.



(a) *barnes*



(b) *radiosity*



(c) *blackscholes*

Figure 33: Heatmap of the memory traffic. The value means the number of packets a corresponding core generates or receives.

of TL ring causes high latency for data packets while the latency of memory data packets on switched network is competitive as the traffic is local.

6.5.1.3 Overall Traffic Pattern

As shown in Figure 34, different applications show various types of traffic patterns. For example, *barnes* exhibits one "hot spot" where a majority of the packets are generated at or destined to, while *radiosity* shows two such hot spots. *blackscholes* shows more balanced traffic distribution across many cores. Along with the traffic, communication pattern shows diversity. Figure 35 displays heatmaps for three applications. Here, value at (x, y) denotes the number of packets generated at y with destination s. While *blackscholes* shows a regular communication pattern between

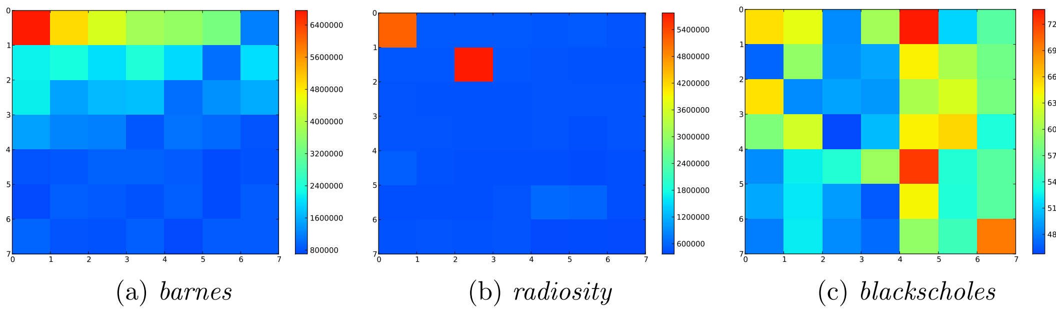


Figure 34: Heatmap of the overall traffic. The value means the number of packets a corresponding core generates or receives.

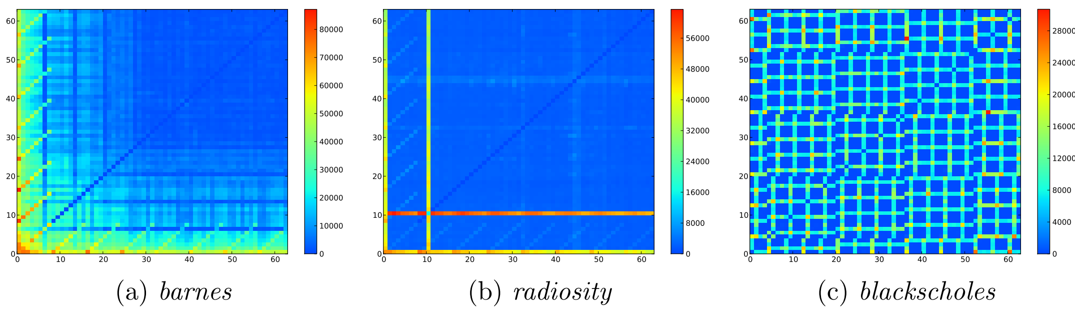


Figure 35: Heatmap of the communication pattern. The value on (x, y) is the number of packets going from y to x.

multiple cores, cores in *radiosity* mainly communicate with two "important" cores.

The traffic pattern significantly affects the effectiveness of hybrid interconnect using TL ring. For example, TL ring's low latency will lose its advantage if most packets are local and show competitive latency even with switched network. Also, TL ring will be overloaded and show less benefit if a majority of packets have to travel long distance. Therefore, it is crucial to correctly identify the representative traffic pattern of various applications.

Figure 36 shows the "average" traffic pattern, which is the distribution of packets for varying packet distances averaged for all tested applications. Along with the distribution, 95% confidence interval is shown. Here, the small range of confidence interval shows that the patterns from various applications do not deviate much, meaning that the average pattern is sufficiently representative. Also, it appears that the

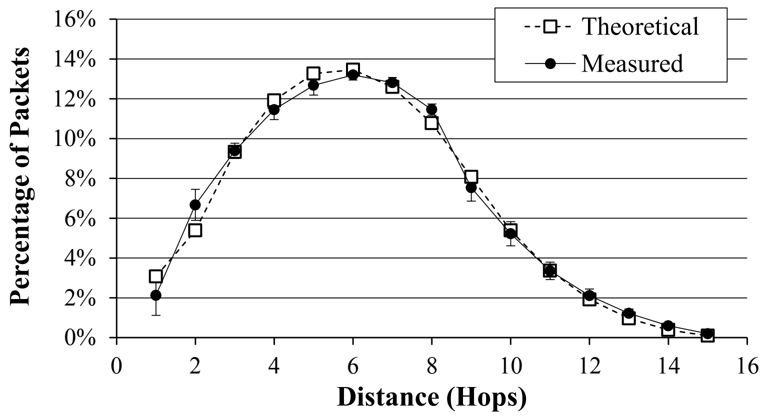


Figure 36: Distribution of traffic for varying packet distance from real applications (Measured). Theoretical random distribution is also depicted (Theoretical).

average traffic pattern closely follows random traffic pattern where the source and destination of the packets are chosen randomly.

Figure 36 also depicts theoretical distribution of the random traffic (*Theoretical*) to verify this finding. The theoretical case lists all possible communication cases among the 64 cores and calculate the distance in hops for each communication. Because there are more combinatorial cases that result in medium-range distances than the short or long distances, the distribution shows skewed bell shape. The random traffic distribution exhibits long tail, denoting a small fraction of packets travel long distance. Therefore, many applications that have representative traffic pattern can be beneficial with hybrid interconnect as the TL ring handles such long-tail packets.

6.5.2 Impact of Latency Reduction with TL

Signal propagation speed in a TL ring is much faster than traditional on-chip wire. As a result, ring-steered packets on average have 55% lower latency than the packets in the baseline mesh interconnect despite serial transmission of bits on a single transmission line and some delays (Figure 37). Furthermore, TL ring provides lower latency when compared with advanced interconnects: ring-steered packets show 44.3% and 43.9% lower latency over concentrated mesh (*Cmesh*) and flattened butterfly (*Flat*), respectively. However, considering that TL ring receives only 13% to 44% of the

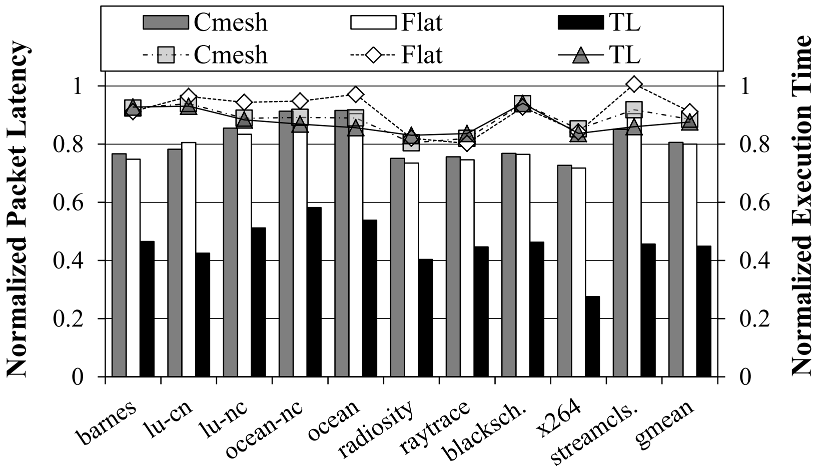


Figure 37: Normalized packet latency to the baseline mesh (left axis) of concentrated mesh (Cmesh), flattened butterfly (Flat) and TL ring with execution time (right axis).

on-chip messages and 2.0% to 9.9% of the traffic (bits) depending on the application, the benefit of overall latency reduction of the ring+mesh interconnect decreases and *Cmesh* and *Flat* provide better overall latency.

However, when I consider performance, the ring+mesh combination outperforms both advanced switched interconnects—ring+mesh provides a 12.4% execution time reduction, compared to 11.5% and 8.9% reduction achieved by using *Cmesh* and *Flat*. This is because my steering scheme concentrates the ring's packet latency reduction to where such reduction is more beneficial. Furthermore, experiments show that the TL ring and steering mechanisms can be gainfully used with these advanced switched interconnects, i.e. a ring+*Cmesh* or ring+*Flat* combination provides a performance benefit beyond that provided by *Cmesh* or *Flat* alone (Section 6.5.5).

6.5.3 Latency Estimation and Re-Steering

My steering scheme is based on latency estimation of mesh and TL ring. Caching-based estimation of mesh latency exhibits less than 30% error for more than 80% of packets in all applications, except in *streamcluster* (where "only" 72% of packets have <30% latency estimation error). For TL latency estimation, I find that 80% of packets can be estimated within 6 cycles of actual latency, except *ocean* and *ocean-nc* where “only” about 75% of packets are estimated within 6 cycles of actual latency. Because the steering scheme employs a threshold that specifies the minimum allowed score (i.e., gap between estimated TL and mesh latency), it can tolerate some estimation error if it still results in steering the packet to the same network (ring or mesh). However, a small portion of packets experience high estimation error and are thus steered to a sub-optimal network, leading to potential loss in performance improvement.

In this experiments, only 0.27% of packets are re-steered on average. *streamcluster* has the most re-steering, 2.3%, which is still rare. These packets wait in the TL queue before finally traveling through the mesh and they suffer a higher latency than they would in the mesh-alone baseline.

Re-steering is affected by how often I check ring-enqueued packets for possible re-steering (period T in Sections 6.3.3). If the period is too short, packets that would have benefited from the ring are send to the mesh after already having accrued a latency penalty by waiting for the ring. If the period is too long, the ring experiences longer periods of saturation (and high overall latencies) during certain message traffic changes. In my experiments, the re-steering period does change how many packets are re-steered, but even with short re-steering periods (10 cycles) the number of re-steered packets is relatively low with 1.7% and the impact on execution time is also marginal (0.3% worse than the optimal period). Interestingly, the results show that the optimal point for the re-steering period (24 cycles) tends to be close to the average packet latency in the mesh interconnect.

6.5.4 Effect of Ring Bandwidth and Steering

Figure 38 shows application execution times with a mesh+ring, normalized to the mesh-only baseline. To evaluate the effectiveness of my traffic steering scheme (shown as *TS*), I also show *RND50* and *RND30* steering, which randomly steer packets to the ring with 50% and 30% probability, respectively. These non-adaptive random

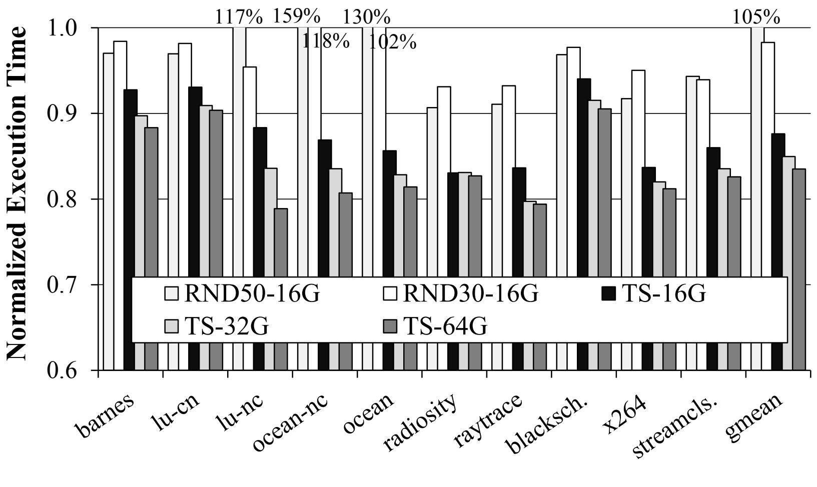
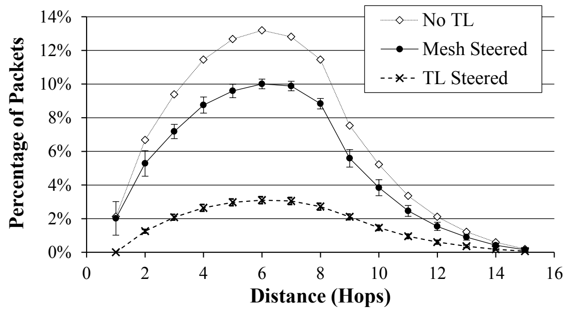


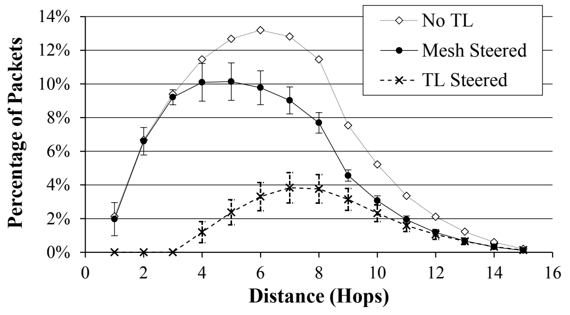
Figure 38: Normalized execution time of traffic steering (TS) with varying TL ring bandwidth, random (RND).

steering approaches provide limited execution time reduction for some applications, mostly those that have less message traffic (e.g., *radiosity*). However, for applications like *ocean-nc*, where on-chip traffic can be significant, sending 50% of packets to TL ring causes too much contention, which results in significant slowdown (59% execution time increase in *ocean-nc*), and even 30% of the traffic proves too much in some applications. On average, *RND30* provides only a 1.7% execution time reduction, and *RND50* results in a 5% execution time *increase.* In some lower-traffic applications, *RND50* provides better performance than *RND30*, because it allows more of the (light) traffic to benefit from the ring's low latency. However, even in these applications, the adaptive steering (*TS*) shows significantly better performance—partly because it allows nearly all traffic to benefit from the TL when traffic is very light, and partly because it is more judicious in selecting which messages to ring-steer during higher-traffic periods.

The adaptive steering sends more packets with large hops to the TL ring as shown in Figure 39b. However, random steering always sends a fixed fraction of packets to the TL ring. It makes TL ring handle more local traffic than the adaptive one (Figure 39a) even if those packets do not benefit from TL ring. It complies with my



(a) Random 50%



(b) Adaptive Steering

Figure 39: Distribution of packets steered to TL ring and switched network with Random Steering (left) and Adaptive Steering (right). Dotted line denotes the distribution of packets with switched network only. 95% confidence interval is shown.

motivation that TL ring manages only the beneficial packets to provide the maximum latency reduction with the limited throughput. Also note that the adaptive steering allows a small fraction of near-traffic packets (e.g. <7 hops) on the TL ring. It is because those packets are esteemed to benefit from using TL ring as the latencies on switched network are likely to be higher with the congestion in the network. Likewise, the long distance packets are not always allowed to use the TL ring in adaptive steering. Simple hop-based steering that directs long distance traffic above a certain distance to the TL ring cannot adapt with such temporal congestion in the networks.

6.5.5 Effect of Advanced Switched Networks

In the combined interconnect, the switched interconnect handles most of the on-chip traffic. While I have used a mesh as a baseline switched interconnect, this approach can be applied to any switched interconnect where there is a significant variation in packet latency depending on the destination. To confirm this, I evaluate my approach using a concentrated mesh (*Cmesh*) and a flattened butterfly (*Flat*) as the baseline. The *Cmesh* I use in this experiments is 4-concentrated (4 cores per switch) and in *Flat*, I halved the link width. These results are shown in Figure 40. Note that the speedups are based on the baseline mesh result. I find that the TL ring provides a

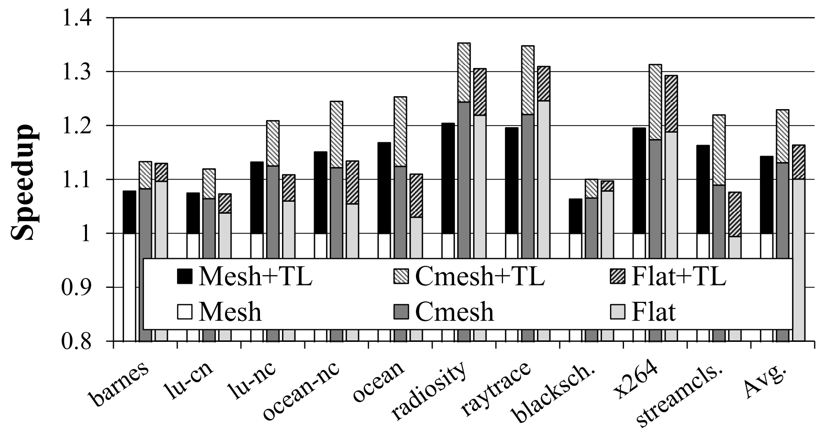


Figure 40: Speedup over baseline mesh when TL ring (16 Gb/s) is used with concentrated mesh network (Cmesh) and flattened butterfly (Flat).

8.7% improvement in speedup (1.23×) over the *Cmesh* (1.13×), and 5.7% improvement (1.16×) over *Flat* (1.10×). These results show that the benefits of adding TL ring are not entirely additive with the benefits of using a more advanced switched interconnect. However, it should be noted that the flattened butterfly is one of the best-performing switched networks that is still suitable for on-chip implementation, so a 5.7% performance improvement in exchange for 1.2% of the chip's metal area may be a good tradeoff that would be difficult to match by further improvements in switched network topology.

6.5.6 Sensitivity to TL Ring Bandwidth

Figure 41 shows average execution time reduction as TL ring bandwidth increases, e.g. by using more than one such ring or more sophisticated signalling. I observe diminishing marginal benefit from each doubling of TL bandwidth, while the costs increase dramatically: a 16 Gb/s TL provides 12.4% execution time reduction; 32 Gb/s yields an additional 2.6% execution time reduction but at 2× the cost, with 64 Gb/s yields another 1.5% improvement but doubles the cost again, etc. This cost-benefit trend confirms my initial insight that one should judiciously use an inexpensive TL, rather than build an expensive TL that can handle all of the on-chip traffic.

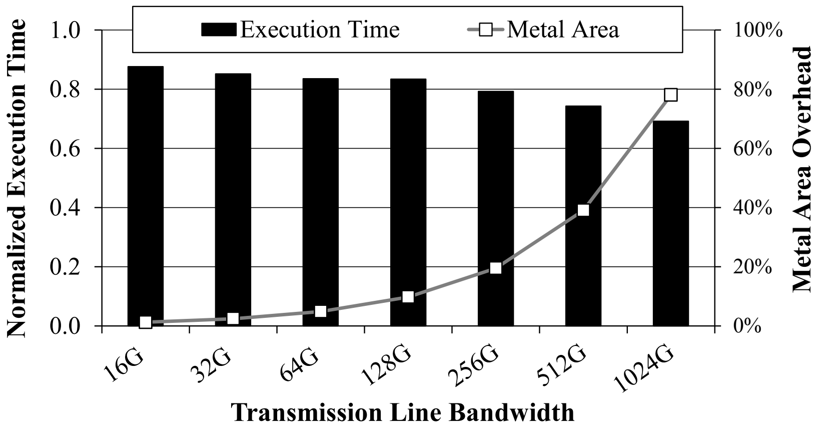


Figure 41: Normalized execution times (averaged for all benchmarks) for configurations with various TL ring bandwidth.

6.5.7 Non-Latency-Critical Applications

Although this evaluation is focused on *latency-critical* applications, my scheme provides some performance improvement (2.9% on average) for *non-latency-critical* applications. Also, my approach does not hurt performance in any applications.

6.5.8 Power Benefit

The main sources of power consumption at TL ring are active components (Table 17). Without any power-hungry switches/routers, the overhead is near constant regardless of the load on the TL ring (0.5W). Therefore, the overall power benefit is equal to the reduction in power consumption on baseline NoC due to the reduced traffic (5.0% on average), minus the (nearly constant) TL ring power overhead.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

7.1 Thesis Summary

This thesis addresses three major problems related to scalability limiters to achieve good performance on multi-core processors: load imbalance, barrier latency and increasing on-chip packet latency. To address each limiter, I devise three approaches that use both software and hardware techniques.

The existence of load imbalance is easy to detect, but it is challenging to determine the cause of the problem and the point in the source code where the problem is introduced. Thus, I propose my first approach called LIME, a framework that automates this process. LIME first uses profiling to collect counts of control flow events and hardware events for the different threads. It then uses clustering and regression to identify a small set of events that introduce significant amounts of imbalance. I show that LIME framework provides accurate and useful feedback to programmers on a set of popular parallel benchmarks.

LIME framework is further extended to handle applications running on heterogeneous processors. Leaving the profiled event counts unchanged, the extension scales execution times to enable LIME analysis. This scaling transforms execution time of threads on big cores as if the threads are run on small cores. Among many scaling methods, I find that average thread cost provides the most accurate result and show that the LIME framework can still operate with heterogeneous cores and identify the source of imbalances accurately.

For the second problem with barrier latency, I present TLSync, a novel hardware barrier implementation that uses the high-frequency part of the spectrum in a transmission-line broadcast network. In contrast to other implementations of hardware barriers, TLSync allows both low barrier latency and multiple thread groups to have their own barrier. This is accomplished by allocating different bands in the radio-frequency spectrum to different groups. Circuit-level and electromagnetic models show that the worst-case latency for a TLSync barrier would be between 4 ns and 10 ns. Given this, the results of the proposed cycle-accurate simulation show that such TLSync barriers provide significant performance improvements for barrier-intensive multi-core applications and improved scalability over software and some hardware-assisted implementations. Moreover, tens of different barriers can be supported using a single physical transmission-line chip-spanning tree while still leaving the transmission line free for baseband (unmodulated) data transmission. Overall, I find that TLSync barrier support is a good solution for the applications suffering barrier performance—TLSync provides scalable and low-latency barrier synchronization for multiple thread groups in a cost efficient manner.

Lastly, I describe an approach where a low-latency unswitched interconnect is synergistically used with a high-throughput switched interconnect in a many-core system. I designed and introduced a ring of transmission line coupler structures with low latency and low throughput. I combined the structures with a low-latency arbitration mechanism that allows efficient use of this ring by many cores. Then, a new adaptive packet steering policy is created to judiciously use the limited throughput available on the low-latency interconnect. Experiments show that the ring+mesh combination adds only 1.3% to the chip area, but provides a 12.4% execution time reduction for parallel applications on average, compared to the mesh alone.

7.2 Limitations and Future Work

Ever since programmers start to design applications that run in parallel, debugging parallel performance has been one of the most important agendas. Techniques and frameworks presented in this thesis address the very tip of a gigantic iceberg of performance debugging and still, there are many issues that need further investigation. Below are some suggestions for the future work.

Firstly, I would like to exploit the load imbalance information identified by LIME framework to further automate the performance debugging process. LIME reports highly accurate information on the cause of load imbalance. In this thesis, I leave the verification of the suspects and repair of the load imbalance to the programmers. Automating this cumbersome process will greatly help application developers. Moreover, LIME framework does post-mortem analysis on the load imbalance: it analyzes imbalance that already occurred. That said, one possible direction for future work is to enable on-the-fly analysis routine that can detect the cause of load imbalance during execution, which would further increase the effectiveness of LIME.

Secondly, I would explore cooperative application of LIME and TL-based hybrid interconnect. The gist of our TL-based hybrid interconnect is the steering logic that makes decision on TL usage. In this thesis, the logic operates based on the local information, i.e., how much latency can be saved for a given packet regardless of the thread that generates it. However, in the global point of view, the decision from the proposed logic may not be an optimal solution. If the packet is from a thread that runs ahead and reaches a global synchronization point first, reducing latency for the packet would not affect the total execution time of the application. LIME can be used together and overcome this local optima problem by providing global criticality information to steering logic through the cause of load imbalance. Note that this collaboration must be preceded by the on-the-fly analysis routine presented in the previous paragraph.

Thirdly and finally, I would like to explore broader application of transmission lines on various synchronization primitives. Although TLSync successfully proposed a fast global synchronization, still, there is much room for transmission lines to implement other synchronization methods using its fast propagation speed. For example, lock can be implemented using ring-shaped TL network and a simple token-based arbitration mechanism. However, the high cost of TL implementation prevents from separately building TL network for each synchronization method. Building a unified TL interconnect that carries multiple types of synchronization signals as well as data packets may solve this cost problem.

7.3 Concluding Remarks

To conclude, the number of available cores in a processor increases rapidly, challenging application developers to write parallel programs that scale well with the increasing cores for better performance. However, writing a scalable application is challenging, resulting many parallel applications to suffer from *performance bugs* caused by *scalability limiters*.

With this thesis, I propose three methods that can help programmers overcome the scalability limiter challenges: (1) LIME, a software framework that addresses load imbalance, (2) TLSync, a hardware support that enables fast on-chip global synchronization and (3) Hybrid interconnect, a hardware technique that supports flagging on-chip communication. To conclude, I hope this thesis can become the baseline direction of how to increase performance of parallel applications for the forthcoming era of many-core processors.

REFERENCES

[1] Abellán, J. L., Fernández, J., and Acacio, M. E., "Efficient and scalable barrier synchronization for many-core cmps," in *Proceedings of the 7th ACM Intl. Conf. on Computing frontiers*, pp. 73–74, 2010.

[2] Advanced Design System, "Agilent Technologies, Santa Clara CA, USA," 2010.

[3] Aguilera, M., Mogul, J., Wiener, J., Reynolds, P., and Muthitacharoen, A., "Performance debugging for distributed systems of black boxes," in *9th Symp. on Operating Systems Principles (SOSP)*, 2003.

[4] Almási, G., Archer, C., Castaños, J. G., Gunnels, J. A., Erway, C. C., Heidelberger, P., Martorell, X., Moreira, J. E., Pinnow, K., Ratterman, J., Steinmacher-Burow, B. D., Gropp, W., and Toonen, B., "Design and implementation of message-passing services for the Blue Gene/L supercomputer," *IBM J. Res. Dev.*, vol. 49, pp. 393–406, 2005.

[5] Arzuaga, E. and Kaeli, D. R., "Quantifying load imbalance on virtualized enterprise servers," in *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, pp. 235–242, 2010.

[6] AWR, "Applied Wave Research, El Segundo, CA, USA," 2010.

[7] Balfour, J. and Dally, W. J., "Design tradeoffs for tiled CMP on-chip networks," in *Proceedings of the 20th annual international conference on Supercomputing*, pp. 187–198, 2006.

[8] Beck, B., Kasten, B., and Thakkar, S., "VLSI assist for a multiprocessor," in *Proceedings of the second Intl. Conf. on Architectual Support for Programming Languages and Operating Systems*, pp. 10–20, 1987.

[9] Beckmann, B. M. and Wood, D. A., "TLC: Transmission Line Caches," in *Proceedings of the 36th annual IEEE/ACM Intl. Symposium on Microarchitecture*, pp. 43–54, 2003.

[10] Beckmann, C. J. and Polychronopoulos, C. D., "Fast barrier synchronization hardware," in *Proceedings of the 1990 ACM/IEEE Conf. on Supercomputing*, pp. 180–189, 1990.

[11] Berner, S., Weber, R., and Keller, R. K., "Enhancing software testing by judicious use of code coverage information," in *Proceedings of the 29th International Conference on Software Engineering*, pp. 612–620, 2007.

[12] Bhattacharjee, A. and Martonosi, M., "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," in *36th Int'l. Symp. on Computer Architecture (ISCA)*, 2009.

[13] Bienia, C., *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, January 2011.

[14] Bienia, C., Kumar, S., and Li, K., "PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," in *2008 Int'l. Symp. on Workload Characterization*, 2008.

[15] Bienia, C., Kumar, S., Singh, J. P., and Li, K., "The PARSEC benchmark suite: Characterization and architectural implications," in *17th Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[16] Borremans, J., Thijs, S., Dehan, M., Mercha, A., and Wambacq, P., "Low-cost feedback-enabled LNAs in 45nm CMOS," in *Proceedings of ESSCIRC '09*, pp. 100–103, 2009.

[17] Bourduas, S. and Zilic, Z., "A hybrid ring/mesh interconnect for network-on-chip using hierarchical rings for global routing," in *Proceedings of the First International Symposium on Networks-on-Chip*, pp. 195–204, 2007.

[18] Cai, Q., González, J., Rakvic, R., Magklis, G., Chaparro, P., and González, A., "Meeting points: using thread criticality to adapt multicore hardware to parallel regions," in *17th Int'l Conf. on Parallel Arch. and Compilation Techniques (PACT)*, 2008.

[19] Carpenter, A., Hu, J., Kocabas, O., Huang, M., and Wu, H., "Enhancing effective throughput for transmission line-based bus," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, pp. 165–176, 2012.

[20] Carpenter, A., Hu, J., Xu, J., Huang, M., and Wu, H., "A case for globally shared-medium on-chip interconnect," in *Proceedings of the 38th annual Intl. Symposium on Computer Architecture*, 2011.

[21] Chang, M.-C. F., Cong, J., Kaplan, A., Liu, C., Naik, M., Premkumar, J., Reinman, G., Socher, E., and Tam, S.-W., "Power reduction of cmp communication networks via rf-interconnects," in *Proceedings of the 41st annual IEEE/ACM Intl. Symposium on Microarchitecture*, pp. 376–387, 2008.

[22] Cheng, L., Muralimanohar, N., Ramani, K., Balasubramonian, R., and Carter, J. B., "Interconnect-aware coherence protocols for chip multiprocessors," in *Proceedings of the 33rd annual International Symposium on Computer Architecture*, pp. 339–351, 2006.

[23] Cianchetti, M. J., Kerekes, J. C., and Albonesi, D. H., "Phastlane: a rapid transit optical routing network," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pp. 441–450, 2009.

[24] Cray Research, Inc., *CRAY T3D System Architecture Overview*, 1993.

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

[26] Dally, W. J. and Towles, B., "Route packets, not wires: on-chip inteconnection networks," in *Proceedings of the 38th annual Design Automation Conference*, pp. 684–689, 2001.

[27] Das, R., Eachempati, S., Mishra, A., Narayanan, V., and Das, C., "Design and evaluation of a hierarchical on-chip interconnect for next-generation CMPs," in *Proceedings of the 15th International Symposium on High Performance Computer Architecture*, pp. 175–186, 2009.

[28] Djordjević, A. R., Napijalo, V. M., Olćan, D. I., and Zajić, A. G., "Wideband multilayer directional coupler with tight coupling and high directivity," *Microwave and Optical Technology Letters*, vol. 54, no. 10, pp. 2261–2267, 2012.

[29] E. M. B. Consortium, "EEMBC benchmark." `www.eembc.org`.

[30] Efroymson, M., "Multiple regression analysis," in *Mathematical Methods for Digital Computers*, pp. 191–203, Wiley, 1960.

[31] Fallin, C., Craik, C., and Mutlu, O., "CHIPPER: A low-complexity bufferless deflection router," in *Proceedings of the 17th International Symposium on High Performance Computer Architecture*, pp. 144–155, 2011.

[32] Fallin, C., Yu, X., Nazario, G., and Mutlu, O., "A high-performance hierarchical ring on-chip interconnect with low-cost routers," *SAFARI Technical Report No. 2011-007*, Sept. 2011.

[33] Galles, M., "Spider: A high-speed network interconnect," *IEEE Micro*, vol. 17, pp. 34–39, 1997.

[34] Gamblin, T., de Supinski, B. R., Schulz, M., Fowler, R., and Reed, D. A., "Scalable load-balance measurement for SPMD codes," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pp. 46:1–46:12, 2008.

[35] Ganguly, A., Chang, K., Deb, S., Pande, P., Belzer, B., and Teuscher, C., "Scalable hybrid wireless network-on-chip architectures for multicore systems," *IEEE Transactions on Computers*, vol. 60, no. 10, pp. 1485–1502, 2011.

[36] Golub, G. H. and Van Loan, C. F., *Matrix computations.* John Hopkins Univ. Press, 1996.

[37] Gottlieb, A., Grishman, R., Kruskal, C. P., McAuliffe, K. P., Rudolph, L., and Snir, M., "The NYU ultracomputer - designing a MIMD, shared-memory parallel machine," in *Proceedings of the 9th Intl. Symposium on Computer Architecture*, pp. 27–42, 1982.

[38] Grecu, C., Pande, P. P., Ivanov, A., and Saleh, R., "Structured interconnect architecture: a solution for the non-scalability of bus-based SoCs," in *Proceedings of the 14th ACM Great Lakes symposium on VLSI*, pp. 192–195, 2004.

[39] Hayenga, M. and Lipasti, M., "The NoX router," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 36–46, 2011.

[40] Hsu, W. T.-Y. and Yew, P.-C., "An effective synchronization network for hot-spot accesses," *ACM Trans. Comput. Syst.*, vol. 10, pp. 167–189, August 1992.

[41] Huck, K. A. and Labarta, J., "Detailed load balance analysis of large scale parallel applications," in *Proceedings of the 39th International Conference on Parallel Processing*, pp. 535–544, 2010.

[42] Intel Corp., "Intel threading building blocks 3.0," 2010. `http://www.intel.com/software/products/tbb/`.

[43] Intl. Technology Roadmap for Semiconductors, "ITRS - 2008 update," 2008. `http://www.itrs.net`.

[44] Intl. Technology Roadmap for Semiconductors, "ITRS - 2009 edition," *http://www.itrs.net*, 2009.

[45] Jain, A. K. and Dubes, R. C., *Algorithms for clustering data.* Prentice-Hall, Inc., 1988.

[46] Jaleel, A., Mattina, M., and Jacob, B., "Last level cache (LLC) performance of data mining workloads on a CMP - a case study of parallel bioinformatics workloads," in *12th Int'l. Symp. on High Performance Computer Architecture (HPCA)*, 2006.

[47] Johns, C. R. and Brokenshire, D. A., "Introduction to the cell broadband engine architecture," *IBM J. Res. Dev.*, vol. 51, no. 5, pp. 503–519, 2007.

[48] Joshi, A., Phansalkar, A., Eeckhout, L., and John, L. K., "Measuring benchmark similarity using inherent program characteristics," *IEEE Trans. Comput.*, vol. 55, no. 6, pp. 769–782, 2006.

[49] Keckler, S. W., Dally, W. J., Maskit, D., Carter, N. P., Chang, A., and Lee, W. S., "Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor," in *Proceedings of the 25th annual Intl. Symposium on Computer architecture*, pp. 306–317, 1998.

[50] Kim, J., Balfour, J., and Dally, W., "Flattened butterfly topology for on-chip networks," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 172–182, 2007.

[51] Kirman, N. and Martínez, J. F., "A power-efficient all-optical on-chip interconnect using wavelength-based oblivious routing," in *Proceedings of the fifteenth Architectural Support for Programming Languages and Operating Systems*, pp. 15–28, 2010.

[52] Kreyszig, E., *Advanced Engineering Mathematics.* John Wiley & Sons, 2010.

[53] Krishna, T., Kumar, A., Chiang, P., Erez, M., and Peh, L.-S., "NoC with near-ideal express virtual channels using global-line communication," *Symposium on High-Performance Interconnects*, pp. 11–20, 2008.

[54] Kumar, A., Peh, L.-S., Kundu, P., and Jha, N. K., "Express virtual channels: towards the ideal interconnection fabric," in *Proceedings of the 34th annual International Symposium on Computer Architecture*, pp. 150–161, 2007.

[55] Kumar, R., Tullsen, D. M., Jouppi, N. P., and Ranganathan, P., "Heterogeneous chip multiprocessors," *Computer*, vol. 38, no. 11, pp. 32–38, 2005.

[56] Kumar, R., Zyuban, V., and Tullsen, D. M., "Interconnections in Multi-Core Architectures: Understanding mechanisms, overheads and scaling," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pp. 408–419, 2005.

[57] Kumar, S., Hughes, C. J., and Nguyen, A., "Carbon: architectural support for fine-grained parallelism on chip multiprocessors," in *34th Int'l. Symp. on Computer Architecture (ISCA)*, 2007.

[58] Larsen, R. and Marx, M., *An Introduction to Mathematical Statistics and Its Applications.* Pearson, 2000.

[59] Leiserson, C. E., Abuhamdeh, Z. S., Douglas, D. C., Feynman, C. R., Ganmukhi, M. N., Hill, J. V., Hillis, D., Kuszmaul, B. C., St. Pierre, M. A., Wells, D. S., Wong, M. C., Yang, S.-W., and Zak, R., "The network architecture of the Connection Machine CM-5," in *Proceedings of the fourth annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 272–285, 1992.

[60] Li, J., Martinez, J. F., and Huang, M. C., "The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors," in *10th Int'l. Symp. on High Perf. Computer Architecture (HPCA)*, 2004.

[61] Lomax, R. G., *Statistical Concepts: A Second Course.* Lawrence Erlbaum Associates, 2007.

[62] Lu, R., Cao, A., and Koh, C.-K., "SAMBA-Bus: A high performance bus architecture for system-on-chips," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 1, pp. 69–79, 2007.

[63] Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V., and Hazelwood, K., "Pin: building customized program analysis tools with dynamic instrumentation," in *SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2005.

[64] Luk, C.-K., Hong, S., and Kim, H., "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 45–55, 2009.

[65] Marty, M. R. and Hill, M. D., "Coherence ordering for ring-based chip multiprocessors," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 309–320, 2006.

[66] Microwave Office, "Applied Wave Research, El Segundo, CA, USA," 2010.

[67] Misurda, J., Clause, J. A., Reed, J. L., Childers, B. R., and Soffa, M. L., "Demand-driven structural testing with dynamic instrumentation," in *Proceedings of the 27th International Conference on Software Engineering*, pp. 156–165, 2005.

[68] Mullins, R., West, A., and Moore, S., "Low-latency virtual-channel routers for on-chip networks," in *Proceedings of the 31st annual International Symposium on Computer Architecture*, pp. 188–197, 2004.

[69] Nanjegowda, R., Hernandez, O., Chapman, B., and Jin, H. H., "Scalability evaluation of barrier algorithms for openmp," in *Proceedings of the 5th Int'l Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, pp. 42–52, 2009.

[70] Norusis, M. J., *PASW Statistics 18 Statistical Procedures Companion.* Pearson, 2010.

[71] NVIDIA, "Tegra 3," 2012. `http://www.nvidia.com/object/tegra-3-processor.html`.

[72] Oh, J., Hughes, C. J., Venkataramani, G., and Prvulovic, M., "LIME: a framework for debugging load imbalance in multi-threaded execution," in *Proceeding of the 33rd International Conference on Software Engineering*, pp. 201–210, 2011.

[73] Oh, J., Prvulovic, M., and Zajic, A., "TLSync: support for multiple fast barriers using on-chip transmission lines," in *Proceeding of the 38th annual International Symposium on Computer Architecture*, pp. 105–116, 2011.

[74] Oncut, A., Badalawa, B., and Fujishima, M., "60GHz-pulse detector based on CMOS nonlinear amplifier," in *IEEE Topical Meeting on Silicon Monolithic Integrated Circuits in RF Systems (SiRF)*, Jan. 2009.

[75] OpenMP Architecture Review Board, "OpenMP application program interface," tech. rep., OpenMP Architecture Review Board, 2010. `http://openmp.org/wp/openmp-specifications/`.

[76] Pan, Y., Kumar, P., Kim, J., Memik, G., Zhang, Y., and Choudhary, A., "Firefly: illuminating future network-on-chip with nanophotonics," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pp. 429–440, 2009.

[77] Peters, T., "Livermore loops coded in c," 1992. `http://www.netlib.org/benchmark/livermorec`.

[78] Phansalkar, A., Joshi, A., and John, L. K., "Analysis of redundancy and application balance in the spec cpu2006 benchmark suite," in *34th Int'l. Symp. on Computer Architecture (ISCA)*, 2007.

[79] Renau, J., Fraguela, B., Tuck, J., Liu, W., Prvulovic, M., Ceze, L., Sarangi, S., Sack, P., Strauss, K., and Montesinos, P., "SESC simulator," January 2005. http://sesc.sourceforge.net.

[80] Sampson, J., Gonzalez, R., Collard, J.-F., Jouppi, N. P., Schlansker, M., and Calder, B., "Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers," in *Proceedings of the 39th Annual IEEE/ACM Int'l Symposium on Microarchitecture*, pp. 235–246, 2006.

[81] Sanderson, C., "Armadillo library," 2010. `http://mloss.org/software/view/176/`.

[82] Santelices, R. and Harrold, M. J., "Efficiently monitoring data-flow test coverage," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 343–352, 2007.

[83] Santelices, R., Jones, J. A., Yu, Y., and Harrold, M. J., "Lightweight fault-localization using multiple coverage types," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 56–66, 2009.

[84] Sartori, J. and Kumar, R., "Low-overhead, high-speed multi-core barrier synchronization.," in *5th Intl. Conf. on High Performance Embedded Architectures and Compilers*, pp. 18–34, 2010.

[85] Schmitz, O., Hampel, S., Orlob, C., Tiebout, M., and Rolfes, I., "Body effect up- and down-conversion mixer circuits for low-voltage ultra-wideband operation," *Analog Integrated Circuits and Signal Processing*, vol. 64, pp. 233–240, 2010.

[86] Scott, S. L., "Synchronization and communication in the T3E multiprocessor," in *Proceedings of the seventh Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 26–36, 1996.

[87] Shang, S. and Hwang, K., "Distributed hardwired barrier synchronization for scalable multiprocessor clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, pp. 591–605, 1995.

[88] Sinha, M., Hsu, S., Alvandpour, A., Burleson, W., Krishnamurthy, R., and Borkar, S., "High-performance and low-voltage sense-amplifier techniques for sub-90nm SRAM," in *Proceedings of the IEEE Intl. Conf. on Systems-on-Chip*, pp. 113–116, 2003.

[89] Sinharoy, B., Kalla, R. N., Tendler, J. M., Eickemeyer, R. J., and Joyner, J. B., "Power5 system microarchitecture," *IBM J. Res. Dev.*, vol. 49, pp. 505–521, 2005.

[90] Srivastava, N. and Banerjee, K., "Performance analysis of carbon nanotube interconnects for VLSI applications," in *Proc. IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, Nov. 2005.

[91] Sun, Y., Jeong, C., Lee, I., Lee, J., and Lee, S., "A 50-300-MHz low power and high linear active RF tracking filter for digital TV tuner ICs," in *2010 IEEE Custom Integrated Circuits Conference (CICC)*, pp. 1–4, 2010.

[92] Tallent, N. and Mellor-Crummey, J., "Effective performance measurement and analysis of multithreaded applications," in *14th SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2009.

[93] Tallent, N., Mellor-Crummey, J., and Porterfield, A., "Analyzing lock contention in multithreaded applications," in *15th SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[94] Tsai, K.-L., Lai, F., Pan, C.-Y., Xiao, D.-S., Tan, H.-J., and Lee, H.-C., "Design of low latency on-chip communication based on hybrid NoC architecture," in *8th IEEE International NEWCAS Conference*, pp. 257–260, 2010.

[95] Udipi, A. N., Muralimanohar, N., and Balasubramonian, R., "Towards scalable, energy-efficient, bus-based on-chip networks," in *Proceedings of the 16th International Symposium on High Performance Computer Architecture*, pp. 1–12, 2010.

[96] Valdes-Garcia, A., Venkatasubramanian, R., Srinivasan, R., Silva-Martinez, J., and Sanchez-Sinencio, E., "A CMOS RF RMS detector for built-in testing of wireless transceivers," in *Proceedings of 23rd IEEE VLSI Test Symposium*, pp. 249–254, 2005.

[97] Vantrease, D., Schreiber, R., Monchiero, M., McLaren, M., Jouppi, N. P., Fiorentino, M., Davis, A., Binkert, N., Beausoleil, R. G., and Ahn, J. H., "Corona: System implications of emerging nanophotonic technology," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pp. 153–164, 2008.

[98] Wolkotte, P., Smit, G., Kavaldjiev, N., Becker, J., and Becker, J., "Energy model of Networks-on-Chip and a Bus," in *Proceedings of International Symposium on System-on-Chip*, pp. 82–85, 2005.

[99] Woo, S., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A., "The SPLASH-2 programs: Characterization and methodological considerations," in *22nd Int'l Symp. on Computer Architecture*, 1995.

[100] Xu, Z., Huang, R., and Bhuyan, L. N., "Load balancing of DNS-based distributed web server systems with page caching," in *Proceedings of the 10th International Conference of Parallel and Distributed Systems*, pp. 587–594, 2004.

[101] Yablonovitch, E., "Can nano-photonic silicon circuits become an intra-chip interconnect technology?," in *Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pp. 309–309, 2007.

[102] Zheng, Y. and Saavedra, C., "Ultra-compact MMIC active bandpass filter with wide tuning range," *Electronics Letters*, vol. 44, no. 6, pp. 424–425, 2008.