ILP AND TLP IN SHARED MEMORY APPLICATIONS: A LIMIT STUDY

A Dissertation

by

EHSAN FATEHI

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

DOCTOR OF PHILOSOPHY

Chair of Committee,	Paul V. Gratz
Committee Members,	Narasimha Reddy
	Sam Palermo
	Riccardo Bettati
Head of Department,	Miroslav M. Begovic

May 2015

Major Subject: Computer Engineering

ABSTRACT

The work in this dissertation explores the limits of Chip-multiprocessors (CMPs) with respect to shared-memory, multi-threaded benchmarks, which will help aid in identifying microarchitectural bottlenecks. This, in turn, will lead to more efficient CMP design.

In the first part we introduce DotSim, a trace-driven toolkit designed to explore the limits of instruction and thread-level scaling and identify microarchitectural bottlenecks in multi-threaded applications. DotSim constructs an instruction-level Data Flow Graph (DFG) from each thread in multi-threaded applications, adjusting for inter-thread dependencies. The DFGs dynamically change depending on the microarchitectural constraints applied. Exploiting these DFGs allows for the easy extraction of the performance upper bound. We perform a case study on modeling the upper-bound performance limits of a processor microarchitecture modeled off a AMD Opteron.

In the second part, we conduct a limit study simultaneously analyzing the two dominant forms of parallelism exploited by modern computer architectures: Instruction Level Parallelism (ILP) and Thread Level Parallelism (TLP). This study gives insight into the upper bounds of performance that future architectures can achieve. Furthermore, it identifies the bottlenecks of emerging workloads. To the best of our knowledge, our work is the first study that combines the two forms of parallelism into one study with modern applications. We evaluate the PARSEC multithreaded benchmark suite using DotSim. We make several contributions describing the high-level behavior of next-generation applications. For example, we show that these applications contain up to a factor of 929X more ILP than what is currently being

extracted from real machines. We then show the effects of breaking the application into increasing numbers of threads (exploiting TLP), instruction window size, realistic branch prediction, realistic memory latency, and thread dependencies on exploitable ILP. Our examination shows that theses benchmarks differ vastly from one another. As a result, we expect that no single, homogeneous, micro-architecture will work optimally for all, arguing for reconfigurable, heterogeneous designs.

In the third part of this thesis, we use our novel simulator DotSim to study the benefits of prefetching shared memory within critical sections. In this chapter we calculate the upper bound of performance under our given constraints. Our intent is to provide motivation for new techniques to exploit the potential benefits of reducing latency of shared memory among threads. We conduct an idealized workload characterization study focusing on the data that is truly shared among threads, using a simplified memory model. We explore the degree of shared memory criticality, and characterize the benefits of being able to use latency reducing techniques to reduce execution time and increase ILP. We find that on average true sharing among benchmarks is quite low compared to overall memory accesses on the critical path and overall program. We also find that truly shared memory between threads does not affect the critical path for the majority of benchmarks, and when it does the impact is less than 1%. Therefore, we conclude that it is not worth exploring latency reducing techniques of truly shared memory within critical sections.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Moore's law [31], which states that the amount of transistors on integrated circuits doubles in a given period, has been the underlying driver of computer advancement for nearly half a century. For many decades, there has been an increase in performance and decrease in power consumption per-transistor as device technology scaled. This phenomenon is known as Dennard scaling [9]. In the last decade, however, the power and performance increase dictated by Moore's Law with Dennard scaling have had diminishing returns. As a result, relying on Moore's law to gain performance has become more difficult due to power constraints. Recently, this trend drove computer architects to chip-multiprocessor (CMP) designs with ever increasing core counts to better leverage these extra transistors. As core counts continue to increase, however, power and performance are unable to proportionally match the previous pace of improvement [12]. Future designs must compensate for inefficient transistor scaling with respect to energy and performance. A characterization analysis of future workloads is imperative in order to ensure that future designs achieve maximum returns in performance with respect to power consumption.

The current industry approach to CMP architecture design is to replicate a single core multiple times. These homogeneous CMPs are expected to be sufficient to run current and near-future applications. This is an ineffective approach to multi-core design, however, as it is only motivated by reducing costs and design effort. Bhadauria et al. showed that current processors are not sufficient for emerging multi-threaded applications [3]. They concluded that current architectures should increase the number of functional units on each core, reduce core size (using in-order execution rather than Out-of-Order execution), and increase core count in order to

improve performance and reduce power.

Typical processor architectures exploit two forms of parallelism in order to achieve scaling performance with increasing transistor density: Instruction Level Parallelism (ILP) and Thread Level Parallelism (TLP). ILP is exploited by architectures that fetch and execute multiple instructions per cycle from a single instruction stream or thread. Maximizing ILP often requires highly complex microarchitectural techniques such as out-of-order (OoO) execution with large instruction windows. These OoO engines consist of hardware managed pools of instructions searched to find ready-to-execute instructions. As a result, ILP exploitation often comes at a high cost in terms of power. Alternately, TLP is exploited by splitting a problem up into multiple threads that can be run simultaneously on more than one processor. Utilizing TLP in typical applications requires that the programmer specify how the problem is partitioned among those threads and defines the exact communication needed between the threads. Scaling performance with processor count requires that TLP applications have a highly balanced load, otherwise overheads will quickly lead to diminishing returns with increasing processor count.

While the processor designs of the 1990's and early 2000's predominantly relied upon ILP exploitation to scale performance, the breakdown of Dennard scaling has driven computer architects towards CMP designs. CMPs integrate many cores onto one die, exploiting TLP to improve performance. TLP exploitation does not preclude ILP exploitation, so having multiple cores on a single die opens up many different design combinations. It is unlikely that a single universal CMP design would be optimal for all applications. A key design challenge lies in determining how to partition chip resources in CMPs between ILP and TLP exploitation. At a high level, chip resources can be spent two ways: increasing the size of each core (for greater ILP exploitation), or by creating additional cores (for greater TLP exploitation).

Understanding when to add cores or change core size is pivotal in optimal multi-core design. The first step is to understand the trade-offs in ILP and TLP in modern applications. To the best of our knowledge there has been no such thorough study in recent years that analyzes such trade-offs in ILP and TLP in modern applications. This gap in knowledge is, in part due to the multi-core era being in its infancy compared to other advancements in computer architecture. While there have been many ILP studies done in the past [35, 44, 25, 16, 34, 2, 6], none of them attempt to understand the relationship between ILP and TLP. Further, none have attempted to answer the question, does TLP exploitation reduce ILP and to what degree.

The main focus of this work is the analysis of next-generation workloads along the axes of ILP and TLP exploitation. The intent is to aid in making more informed CMP architectural design decisions. Generally, increasing core size (by increasing cache size, instruction window size, the number of functional units, etc.) results in an increase in ILP extraction; while increasing core count results in an increase in TLP extraction. However, for applications with imbalanced loads, a heterogeneous CMP design composed of a few high ILP cores, and many of low ILP cores (for TLP extraction) might achieve higher overall efficiency. An example of this architecture is shown in Figure 1.1. Our limit study is aimed at narrowing the design choices available by giving guidelines on how future micro-architectures should be designed. We begin by conducting a ground-up workload characterization analysis.

Current cycle-level simulation tools provide insight into application performance on current microarchitectures. However, the limitations of cycle-level simulation prevent exploration of application instruction and thread-level scaling properties necessary to drive future transformational microarchitectural designs. There is a pressing need for a simulator capable of studying the scaling of shared-memory, multi-threaded benchmarks in the limit. Thus, we created DotSim in order to fulfill

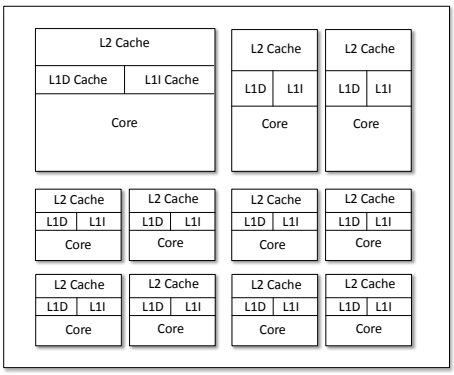


Figure 1.1: Example of a multi-core heterogeneous architecture.

the needs of identifying bottlenecks in architecture design.

DotSim is a trace-driven toolkit designed to explore the limits of instruction and thread-level scaling and identify microarchitectural bottlenecks in multi-threaded applications. DotSim constructs an instruction-level Data Flow Graph (DFG) from each thread in multi-threaded applications, adjusting for inter-thread dependencies. The DFGs dynamically change depending on the microarchitectural constraints applied. Exploiting these DFGs allows for the easy extraction of the performance upper bound. DotSim is discussed thoroughly in Chapter 2.

In Chapter 3, we perform an ILP and TLP limit study on emerging multi-threaded workloads. We use DotSim to construct an instruction-level Data Flow Graph (DFG) from each thread in multi-threaded applications that includes inter-thread dependencies. Using this simulator we evaluate ILP and TLP using the PARSEC shared memory multiprocessor benchmark suite [4]. These experiments determine how TLP extraction affects the ILP availability, how ILP is affected by window size, and the affects of ILP on thread dependencies. We also study the critical path (CP) of each benchmark with the goal of studying both the thread-level load balance and the instruction-level parallelism of the CP segments, thus indicating how wide a core

must be to achieve a given calculated performance.

We extend our limit study in Chapter 3 to explore the benefits of prefetching shared memory within critical sections. We we use our novel simulator DotSim to conduct an idealized workload characterization study, focusing on the data that is truly shared among threads, using a simplified memory model. We explore the degree of shared memory criticality, and characterize the benefits of being able to use latency reducing techniques to reduce execution time and increase ILP. We find that on average true sharing among benchmarks is quite low compared to overall memory accesses on the critical path, and overall program. We also find that truly shared memory between threads does not affect the critical path for the majority of benchmarks and when it does the impact is less than 1%. Therefore, we conclude that it is not worth exploring latency reducing techniques of truly shared memory within critical sections.

1.1 Thesis Statement

This dissertation proposes microarchitecture design is far from optimal, and that conducting a limit study will help aid in optimizing future chip design. By identifying bottlenecks in microarchitecture designs and determining an upper bound limit on performance, computer architects can then make better informed design decisions when it comes to building CMPs . In this thesis, we conduct a limit study simultaneously analyzing the two dominant forms of parallelism exploited by modern computer architectures: Instruction Level Parallelism (ILP) and Thread Level Parallelism (TLP). This study gives insights into the upper bounds of performance that future architectures can achieve. Furthermore it identifies the bottlenecks of emerging workloads. To the best of our knowledge, our work is the first study that combines the two forms of parallelism into one study with modern applications. We

evaluate the PARSEC multithreaded benchmark suite using our specialized trace-driven simulator called DotSim.

1.2 Dissertation Contributions

The **first contribution** of this thesis, is our open source trace-driven simulator DotSim. DotSim, is an abstract microarchitectural simulator that identifies bottlenecks in multi-threaded programs. DotSim design and infrastructure, allows it to excel in first-order modeling of novel microarchitectural approaches. Also, due to its simplicity it can be used to validate other simulators by providing an upper bound on performance.

We make the following contributions with DotSim:

1. *We develop DotSim, an abstract microarchitectural simulator. DotSim uses multi-threaded program traces as inputs and converts them into DFGs. These DFGs enable the determination of the critical path and execution time of applications under arbitrary resource constraints.*

2. *DotSim excels at first-order modeling of many novel microarchitectural approaches, such as memory synchronization speculation. This modeling can be in the form of higher levels of abstraction, allowing the exploration of novel microarchitectural approaches without a particular concrete design for implementation. Thus saving time and effort in early design exploration.*

3. *Due to its simplicity DotSim can be used to validate other simulators, particularly execution-driven simulators, by putting an upper bound on performance.*

The **second contribution** of this thesis, we use DotSim to conduct a limit study on multithreaded applications. This limit study conducts seven experiments exploring the relationship of of ILP and TLP in next generation benchmarks. These experi-

ments involve breaking the application into increasing numbers of threads (exploiting TLP), observing the impact instruction window size, realistic branch prediction, realistic memory latency, and thread dependencies on exploitable ILP with respect to performance. We then provide recommendations on future architecture design based on our results.

Our experiments explore the demands of these applications with respect to hardware architectures, such as exploring the effects of realistic branch prediction and memory latency. Analysis of these multi-threaded applications provide insight on trade-offs in multi-core designs. This work answers the following seven questions:

1. *What is the upper bound on ILP?* We provide quantitative data on how much ILP headroom is available in modern applications in relation to how much is currently extracted. We find that current architectures are far removed from the ILP limits found using our methodology. For example, current machine ILP can differ by as much as 929x versus a processor with infinite resources.

2. *What is the threading inefficiency of each benchmark?* We quantitatively explore workload imbalance in these applications. Understanding the load imbalance will help identify bottlenecks in the microarchitectural design. With increasing TLP the load imbalance increases, but we find the rate of the inefficiency depends greatly on the complexity of the particular benchmark.

3. *What is the impact on ILP as we scale cores?* We quantify the actual trade-off in TLP and ILP for these applications. This will help determine whether it is better to increase core size or to add more cores. We find, generally, increasing TLP extraction does indeed affect ILP. The relation between the two is however, highly dependent on the benchmark.

4. *What is the impact on ILP when imposing instruction window size restrictions?* We examine how ILP increases with window size for these applications. We find that there is large amount of ILP that is not being exploited within a 128 to 512 instruction distance. Often more than 10x the amount found in real machines. The majority of ILP, however, is much further than 5000 instructions away making it unlikely a traditional instruction window will be able to capture the majority of ILP available.

5. *What is the effect of thread dependencies on ILP?* We attempt to quantify the impact that high level thread dependencies have on performance reduction. We find that all but one benchmark's performance was significantly impacted by thread dependencies. As core count increases, the performance reduction caused by thread dependencies increases. Although, the performance reduction occurs at different rates dependent on the application.

6. *What is the impact on ILP when imposing realistic memory system latency and branch prediction accuracy restrictions?* We examine how ILP is affected when imposing realistic memory latency and branch prediction for these applications. We find that there is on average a 31% reduction in ILP when adding realistic memory latency model depending on the benchmark, and on average 67% reduction when adding realistic branch prediction. Overall, branch prediction is the stronger bottleneck than memory latency under these constraints. These results provide motivation to put more effort into improving branch prediction, despite the field being very mature.

7. *What are the critical path's thread composition characteristics?* We attempt to provide an understanding the critical path characteristics with the perspective of gaining performance and reducing power. We find the composition of the

critical path to be interesting and potentially exploitable.

With respect to the experiments listed above we found that no benchmark reacted in a similar manner for all seven questions. This suggests that an optimal multi-core design (with respect to power and performance) is highly dependent on the application running on it, arguing for a dynamic and heterogeneous design. Our examinations concluded that the multithreaded benchmarks differed vastly from one another. As a result, we expect no single, homogeneous, micro-architecture will work optimally for all, arguing for reconfigurable, heterogeneous designs.

The **third contribution** we conduct an idealized workload characterization study, focusing on the data that is truly shared among threads, using a simplified memory model. Here we quantify the amount of true sharing done by threads in multi-threaded benchmarks, as well as perform impact analysis of reducing latency of memory shared among threads.

We make the following contributions with our idealistic shared-memory workload characterization analysis:

1. *The show the amount of true sharing done among threads is trivial except for one of six benchmark where sharing represented 20% of all memory accesses .*

2. *We show the truly shared memory between threads does not affect the critical path. Therefore has on average, a minimal impact on execution time, with an upper bound of increasing performance by less than <1%*

3. *We show that there is not an exploitable sharing patterns between benchmarks to gain performance.*

2. DOTSIM: A TRACE-DRIVEN SIMULATION TOOL KIT

In this chapter, we introduce DotSim, a trace-driven simulation tool kit for shared-memory, multi-threaded application analysis. DotSim is designed to explore the relationship between ILP and TLP. Conducting such a study requires the use of a specialized, trace-driven simulator. No existing simulator is capable of conducting such a limit study utilizing modern multi-threaded applications exists. Further, no simulator exists that can explore the bounds of scaling along several axes while adding or removing arbitrary constraints. To accomplish this task, DotSim constructs an instruction-level Data Flow Graph (DFG) from each thread in multi-threaded applications. These DFGs are then stitched together at the application level by recognizing and adjusting for inter-thread dependencies through memory and via synchronization semantics.

2.1 Why DotSim

Most microarchitecture simulators are either execution driven or trace driven. Execution driven simulators are typically cycle accurate (*ie.* they provide an approximation of performance in terms of the cycles it takes for a given application to execute), which enables these simulators to model performance and behavior quite accurately. This accuracy is driven by detailed, microarchitectural level modeling of the many and varied structures of a microprocessor. Detailed, low-level modeling comes, however, at a cost of simulator implementation complexity. Modeling new microarchitectural features in an execution driven simulator often requires many man-months of effort in implementation and tuning.

Trace-driven simulators, are fundamentally different as they require pre-executed dynamic instruction stream traces for off-line analysis. Therefore, they rely on real

machines or execution driven simulators to generate these traces. The benefit of this approach is that these traces have control flow, thread synchronization lock order, and memory disambiguation already known, dramatically reducing simulator implementation complexity and simulation time. With the dynamic instruction flow determined, the upper-bound performance can be determined via analysis under the assumption of no resource constraints (*eg.* unlimited instruction window, functional units, etc.), thus enabling easier identification of microarchitecture bottlenecks. Additionally, the complexity of implementing new microarchitectural design ideas is dramatically reduced. Recent work argues that complexity of many execution driven simulators mentioned above can often lead to hard-to-debug performance bugs [33, 38]. Trace-driven simulators elide this complexity and thus may more easily produce relatively accurate performance bounds estimates.

DotSim is designed with the goal of simulating processor microarchitectures with varying degrees of abstraction. As conducting limit studies can be incredibly resource and time intensive, we made DotSim as simple and resource light as possible. While existing cycle-level simulation tools can provide insight into application performance on current microarchitectures, the limitations of cycle-level simulation prevent exploration of application instruction-level and thread-level scaling properties necessary to drive future transformational microarchitectural designs. To the best of our knowledge there is no existing simulator capable of studying the ILP and TLP scaling limits of shared-memory, multi-threaded benchmarks. DotSim is such a tool with this goal.

2.2 Related Work

DotSim, is rather unique in terms of its ability simulate with varying degrees of fine-grain abstraction at the microarchitecture level. As a result of its varying degree

of flexibility in simulation, it has the potential for broad use in conducting research. Due to the broad spectrum DotSim touches on, we organize the related work in three different categories: abstract, trace-driven and detailed simulators. For each subsection we compare and contrast past work to DotSim.

2.2.1 Abstract Simulators

Abstraction models are used to study future architectures which are currently difficult to implement in a detailed fashion. Abstraction models are often the first step in exploring a new, non-trivial idea.

DotSim is most directly influenced by the study published by Hill et al. [19]. They derived simple mathematical models to study multicore topologies and trade-offs in terms of ILP and TLP. Their study was done at the highest level of abstraction, a pure mathematical model, and thus leaves many open questions. In many respects, DotSim was designed to provide answers to the questions posed by Hill et al., providing quantification of TLP and ILP tradeoffs.

Guz et al. [17], developed mathematical models to study the effects of caching versus multi-threading. Their goal was to provide an in-depth understanding of the memory wall problem. In a second paper, Guz et al. [18] continued work in studying the limits of architecture by modeling the tradeoffs between Many-Core machines and Many-Thread machines. DotSim, is designed for studying, identifying and quantifying bottlenecks in multi-parallel applications including the two studies conducted above. DotSim, will likely provide more accurate results, at a cost of simulator complexity.

Esmaeilzadeh et al. [12] modeled multicore scaling limits as a factor of device scaling in order to measuring speed up of parallel applications for the next five transistor generations. Esmaeilzadeh et al. used a detailed performance model of

upper-bound performance. DotSim, was designed to more directly calculate the upper-bound performance limits, without requiring any mathematical modeling.

The abstract and mathematical models used in architecture exploration have the benefit of being simple to implement and require little compute effort, however, these benefits come at the cost of relatively low accuracy. DotSim, and other trace-driven simulators, generally show higher accuracy, though the compute time and complexity are also higher.

2.2.2 Trace-Driven Simulators

Very few simulators currently exist that have similar characteristics, methodology and purpose like DotSim. MaxPar [23], developed in the mid 1980s, analyzes data and instruction dependencies in parallel systems. MaxPar is designed to measure inherent parallelism in applications, identifying microarchitectural bottlenecks, as does DotSim. MaxPar, however, was not designed to execute current benchmarks which include inter-thread dependencies. Rico et al. created a methodology to handle flexible trace-driven multi-threaded simulations [38]. DotSim and Rico et al.'s methodologies are similar in that they support parallel traces. However, Rico et al.'s methodology is designed to reduce computation time, and focus on scheduling and managing parallelism techniques with respect to hardware. Our simulator supports research in scheduling and parallelism techniques. Finally, DotSim supports shared memory multithreaded applications, specifically Pthreads which is not supported by Rico et al.'s methodology.

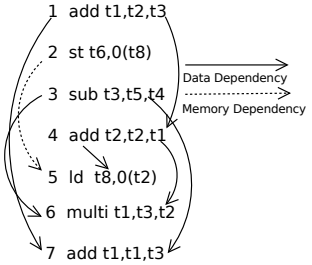
Monichero et al. [30] proposed a novel methodology to emulate a simulator that can support hundreds of cores. Their methodology is similar to ours as we will describe in Section 2.4. Their methodology allows for these traces to emulate up to 1000+ cores, by identifying traces via threads and then pinning them to new

simulated cores.

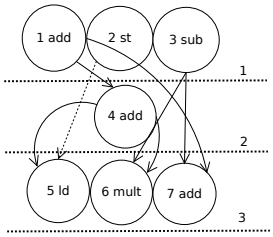
2.2.3 Detailed Simulators

In this section we split detailed simulators into two categories, cycle-level execution driven simulators, and profiling tools. We view cycle-accurate execution driven simulators [5, 45, 26, 39, 13], as complimentary to DotSim, in that they are often the final step in microarchitectural modeling before implementation. DotSim was designed to identify bottlenecks in multi-threaded microarchitectures and determine the limits of performance under new and novel microarchitectural techniques at an earlier stage in development. This task is very difficult to perform with typical cycle-level simulators (see Section 3.1). However, due to DotSim's ability to change the degree of hardware abstraction, DotSim can be the first step in identifying the potential benefit of an idea, prior to using an execution driven-simulator. Often this first order modeling may be sufficient in measuring the benefit of an idea. Furthermore, first order modeling may be the best solution to measure performance as it can provide a reduction in noise such as no operating system interference, or unknown bugs introduced accidentally due complexity of creating a detailed simulator [33].

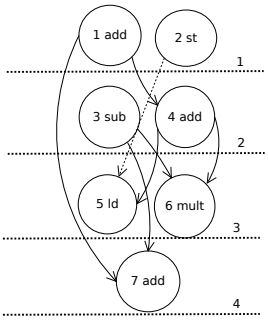
The second category are profiling tools that use the underlying native machine to gather statistics [32, 20, 27]. This often requires knowledge of the underlying benchmarks, to insert code so they can be properly profiled. In DotSim, there is no requirement to understand the inner working of the benchmark under test, thus saving time and extra work. Most importantly, analysis of these benchmarks with profiling tools is only limited to experimenting with the native machine conducting the tests. DotSim is not restrictive as it can replicate a broad spectrum of computer architecture.



(a) Assembly fragment with dependencies highlighted, with Dot language translation.



(b) Unconstrained Data-flow graph (DFG) for code in part (a).



(c) Constrained DFG for code in part (a).

```
1 0-0|I
2 0-1|S|0xFFA
3 0-2|I
4 0-3|I
  0-0|I>0-3|I
5 0-4|L|0xFFA
  0-3|I>0-4|L
  0-1|S>0-4|L
6 0-5|M
  0-2|I>0-5|M
  0-3|I>0-5|M
7 0-6|I
  0-0|I>0-6|I
  0-2|I>0-6|I
```

(d) DotSim trace language for code in part (a).

Figure 2.1: Data-flow graphs (DFGs) for assembly fragment.

2.3 DotSim Overview

As discussed in Section 3.1, DotSim's primary goal is to calculate the limits of scaling in real multi-threaded applications. DotSim has a unique ability to explore the bounds of scaling in shared-memory, multi-threaded applications along several axes while adding or removing arbitrary constraints. DotSim constructs a data-flow graph (DFG) from the dynamic instruction stream trace of the program's execution. This DFG is a directed, acyclic graph consisting of nodes, edges and edge weights,

where nodes represent instructions, edges represent dependencies and edge weights represent the latency of dependency resolution. Using DFGs to conduct architecture limit studies is not new. A similar approaches were taken by prior work [2, 23]. However, we add support for intra-thread dependencies through synchronization primitives and through-memory dependencies.

To create a trace that can be used to generate a DFG, it must first be preprocessed. The preprocessing translates a simple trace of the dynamic instruction stream (*ie.* the instructions actually executed, in order of execution), into nodes (instructions), edges (dependencies) and edge weights (functional unit latencies). We note that after preprocessing, the trace is effectively ISA independent, thus porting any given ISA's instruction stream trace only requires porting this preprocessing component. During this preprocessing stage all dependencies among instructions both intra-thread (through the register file and memory) and inter-thread (through memory) are resolved. Identifying these dependencies prior to simulation allows simulation to proceed more quickly. These preprocessed traces are then fed to the DFG generation stage, where a dynamic Data Flow Graph (DFG) for each thread is constructed.

In the remainder of this section, we discuss single threaded and multi-threaded DFG generation and processing.

2.3.1 Single-Threaded DFGs

DotSim creates a DFG in which instructions (nodes) are interconnected via directed, weighted edges (dependencies). The weight of each edge represents the latency of the production and transmission of operand from producing the instruction to the consuming instruction. The DFG is dynamically adjusted based on microarchitectural constraints given to DotSim. Each edge's weight represents cycle time,

therefore altering or removing constraints will change the shape of the DFG (height and width). DFGs makes measuring performance metrics, such as ILP, trivial. The height of the tree is the effective cycle count required to execute the program, given the arbitrary resource constraints. These arbitrary resource constraints, such as issue width or cache latency, can be changed individually or in concert, thus enabling the identification of microarchitectural bottlenecks.

Figure 3.1a shows a simple assembly pseudo-code fragment with register data dependencies highlighted with solid lines, and memory data dependencies shown with dashed lines. In this example, the effective address of the store (st) and load (ld) alias to the same memory location forming a true data dependence through memory. In this example, we assume an ideal processor core model with infinite physical registers, perfect branch prediction, perfect memory-address aliasing (addresses are known, thus unrelated loads can move in front of stores). Further, we assume unlimited hardware resources such an infinite instruction window, functional units, physical registers, and single cycle memory latency. Figure 3.1b shows the corresponding unconstrained DFG, as created by DotSim. The nodes represent instructions, and the edges represent producer-consumer data dependencies among the instructions. For the purpose of this limit study, all instructions are assumed to take one cycle, thus the edge weights are all assumed to be 1 and are not shown. The maximum height of this DFG, 3 cycles, represents the number of cycles this code fragment would require for execution in an ideal machine.

Figure 2.1c, shows another DFG for the same code fragment shown in Figure 3.1a. In this second example, the instruction window has been constrained to a width of two instructions, producing the DFG shown. Note the height of the DFG has changed to 4, and the overall max width is now two.

ILP Calculation: ILP is calculated from the generated DFG. Here we define the

average ILP of a given single thread (ILP_{ST_AVG}) to be the average number of instructions that can be executed under the given machine constraints. Equation(3.1), is used to calculate ILP_{ST_AVG}.

$$ILP_{ST\_AVG} = \frac{I_{all}}{H} \tag{2.1}$$

In this equation, I_{all} is the total number of instructions in the DFG, and H is the height of the DFG, representing cycle count of the ideal machine for simplicity. Thus, for Figure 3.1b, which shows a DFG for the code in Figure 3.1a with unlimited resources, the ILP_{ST_AVG} is $\frac{7}{3} = 2.33$. Figure 2.1c shows another DFG for the same code, however with a window size constraint of size 2. In this figure, the ILP_{ST_AVG} is $\frac{7}{4} = 1.75$. The ILP_{ST_AVG} is affected by imposing a window size constraint of 2, therefore showing that under this simple constraint, window size is a bottleneck.

2.3.2 Multi-Threaded DFGs

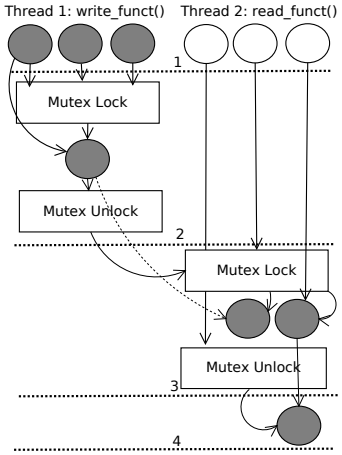
Calculating ILP of shared memory multi-threaded applications using DFGs requires identifying intra-thread dependencies. Intra-thread dependencies occur via thread synchronization constructs as well as through store to load producer-consumer relationships. We recognize these intra-thread dependencies in two ways. First, to preserve correctness, true dependencies through memory caused by communicating load-store pairs between threads are modeled as edges by DotSim. Second, DotSim identifies and captures synchronization constructs (*e.g.* locks, barriers, etc.), and models these constructs as dependencies between threads.

Figure 2.2a illustrates an example multi-threaded program fragment containing a mutex thread synchronization construct. In this example, the two parallel functions, "$write\_funct()$" and "$read\_funct()$", executessimultaneously in two different threads. Here "$write\_funct()$" writes to parts of the shared array $x[]$,

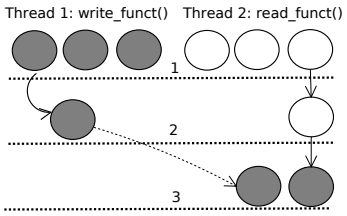
```
void write_funct(){
....
...
while(loop)
....
y[j]++;
pthread_mutex_lock (&lock);
x[j]=y[j]*n;
pthread_mutex_unlock(&lock);
}

void read_funct(){
....
...
while(loop)
....
j++;
pthread_mutex_lock (&lock);
c[j]=x[j];
pthread_mutex_unlock(&lock);
}
```

(a) Multi-Threaded pseudo-code fragment.



(b) DFG for code in part (a) assuming two concurrent threads. Both in-memory dependencies and thread synchronization constructs honored.



(c) DFG for code in part (a) with thread synchronization constructs removed.

Figure 2.2: Multi-threaded code fragment and associated DFG, with and without thread synchronization constructs honored. Darkened nodes represent instructions along the critical path.

while "$read\_funct()$" reads the array. Access to the array is synchronized by the $pthreads\_mutex\_lock()$ to ensure correctness. Figure 2.2b shows the DFG for these two threads with the thread synchronizing mutex lock in place.

In this example, the height of Thread 1's DFG (H_1) is two. Thread 2's DFG (H_2) is four because it inherits Thread 1's height after the mutex unlock (since it has the greater H of the two at this synchronization point). As per Equation (3.1), the ILP_{ST_AVG} of Thread 1 is 2 (The total number of instructions in Thread 1, I_{all1} is 4) and for Thread 2 is 1.5 (I_{all2}=6). These results are averaged across threads to calculate the average ILP of multi-threaded (MT) application DFGs following Equation (3.2).

$$ILP_{MT\_AVG} = \frac{\sum_1^N (I_{alln}/H_n)}{N} \tag{2.2}$$

Here H_n, is the height of thread n's DFG, for threads 1 to N where N is the total number of threads in the benchmark. I_{alln} is the total number of instructions in a given thread. For the DFG shown in Figure 2.2b, the ILP_{MT_AVG} is $\frac{2+1.5}{2} = 1.75$.

Multi-threaded benchmarks bring additional complexity when measuring performance, since all threads depend on each other. Prior work shows that metrics such as average ILP often do not provide a full picture of application performance and scaling [14]. To address these difficulties, we introduce a critical path (CP) ILP metric. Figure 2.2b illustrates DotSim's CP metric. In the figure, a subset of nodes are highlighted grey, these are nodes that lie on the application's height defining critical path. Note, that the figure shows portions of both threads have an impact on the lower-bound limits on execution time. Thus, both threads determine the overall height (H_{max}) of this simplified multi-threaded example. Here, speeding up execution in Thread 2 would not lead to a significant performance increase, due to Thread 1 acquiring the mutex lock first. Although Thread 2 has a higher height after the lock, it inherits the height of 2 from Thread 1 after the release of the mutex. Thus both Thread 1 and Thread 2 have an impact on the overall H_{max} of the program. Instructions from Thread 2 prior to the lock and instructions from Thread 1 after the lock form the *critical path* (CP) of the application. We formally define the program's CP as the dependency chain of *thread segments* through the program that determines the H_{max}, *ie.* the execution time of the program. Much insight can be extracted from per-thread ILP as well as the critical path ILP. For the DFG in Figure 2.2b, the number of instructions on the CP is 7, and the CP takes 4 cycles to execute, therefore we define the ILP_{MT_CP}=1.75 under the given constraints. More

formally ILP_{MT_CP} is calculated as shown in Equation (3.3).

$$ILP_{MT\_CP} = \frac{\sum_1^k I_{kn}}{H_{max}} \tag{2.3}$$

In this equation, I_{kn} is the segment of instructions that are under the height (H_{max}) defining segments of each thread in the DFG.

We note that creating a CP metric is a challenging and somewhat fraught question, with several possible derivations. After careful study and consideration, we chose the current CP metric because it provides more insight than the alternative approaches. Specifically, the current metric captures the average dynamic ILP width of the CP segment in question. Therefore, giving the required width of a core, in order to achieve the performance shown.

A unique feature of DotSim is its ability to explore the limits of TLP by removing all inter-thread synchronization constructs from the code (e.g. Locks, Barriers, etc.), while preserving true inter-thread data dependencies through the memory system to ensure correctness. This feature allows the researcher to explore the ideal performance limit that techniques such as lock and barrier speculation/elision might yield [36, 28]. Figure 2.2c illustrates this feature. In the figure, a DFG is reconstructed from the code in Figure 2.2a after the removal of inter-thread synchronization constructs. Note, correctness is ensured by continuing to enforce direct, producer/consumer relationships between threads through memory (store-to-load communication between threads). This can be compared against the DFG with synchronization constructs intact (Figure 2.2b) to explore the speedup that thread synchronization speculation could achieve in the limit. For the DFG in Figure 2.2c, the number of instructions on the CP is 6, and the CP takes 3 cycles to execute, therefore we derive ILP_{MT_CP}=2, under the given constraints after removing syn-

chronization constructs.

2.4 DotSim Implementation

DotSim is implemented in five stages, where the first four stages need only be performed once for a given dynamic instruction stream trace. A single dynamic instruction stream trace is required for each examined number of threads, N_{min}. In our implementation, each stage is a separate linux process and thus may be chained together simultaneously with unix *pipes*, or may be run serially. Figure 2.3 shows all five stages, from *Trace Generation* to *DFG Processing*. In this section we first describe the DotSim trace file format. We then describe the four *Preprocessing Stages*, explaining in detail the goal and responsibility of each stage. The *DFG Processing* stage is later examined in detail.

2.4.1 DotSim Trace Language Format

In order to simplify and speed up subsequent DFG processing, we developed a concise trace syntax for expressing the relationships and dependencies between instructions. While it would be possible to directly generate a DFG from an unprocessed dynamic instruction stream, this approach would require a significant redesign of the DFG generation code in order to support new ISAs. Further, implementing our own trace language allows us to offload several one-time tasks in trace preprocessing, reducing the complexity of the DFG Processing stage, as we will show.

DotSim's trace language syntax is loosely inspired by the Dot language [10]. In the DotSim trace language each line represents either a node (instruction) or an edge (dependency). Instruction node lines have the following syntax:

$ThreadId\text{-}InstrNum|InstrClass|EffAddr$

Where $ThreadId$ is the application thread id, $InstrNum$ is the sequential instruction number within that thread, $InstrClass$ is the instruction class (*eg.* "S" for store,

or "A" for integer arithmetic), and $EffAddr$ is the effective address (only used for load and store instructions).

Edges in the DotSim trace language take the following form:

$ThreadId_1$-$InstrNum_1|InstrClass_1$ >

$ThreadId_2$-$InstrNum_2|InstrClass_2$

Representing instruction $ThreadId_2$-$InstrNum_2|InstrClass_2$ being dependent upon an operand from $ThreadId_1$-$InstrNum_1|InstrClass_1$, either through the register file or through memory. Note that edge lines must come sequentially after the node line for the dependent instruction in the trace file.

Figure 2.1d shows the DotSim trace language code for the instruction stream in Figure 3.1a.

2.4.2 Preprocessing Stages

The four *Preprocessing Stages* collectively make up the Front End of the DotSim toolkit. Typically, the first three stages are executed simultaneously, using linux named pipes to chain the output of one stage to the input of another. Using pipes is not mandatory, instead it is done to improve simulator execution time by reducing the required number of accesses to hard drive storage. We note that since DotSim is a composed of a modular set of individual programs, one for each stage, it is trivial to replace one or more stages with user defined components.

2.4.2.1 Trace Generation

DotSim's first stage, *Trace Generation*, consists of a dynamic instruction stream trace capture from either a binary instrumentation tool or an architectural-level simulator. In the initial DotSim implementation, the Trace Generation stage was built leveraging gem5's [5] simple-atomic, functional simulation model for the Alpha ISA. In generating dynamic instruction stream traces from gem5, several challenges had

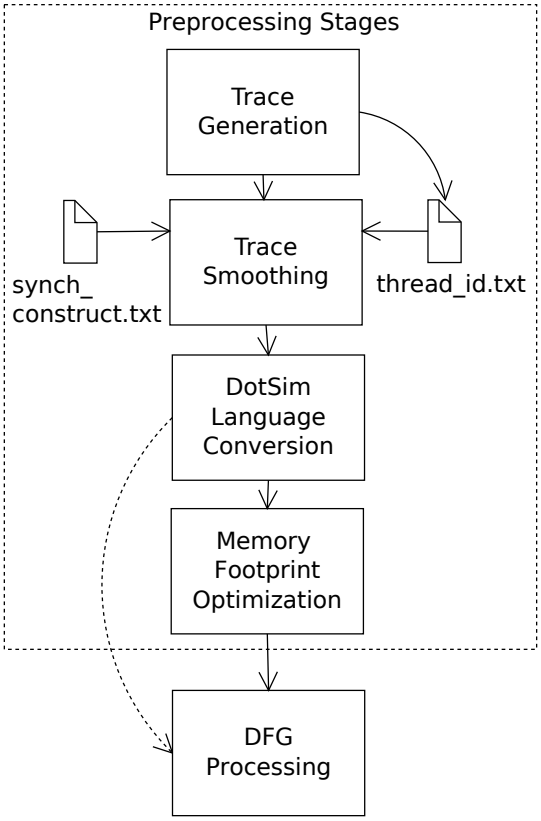


Figure 2.3: The 5 stages of DotSim. Note Memory Management stage is optional.

to be overcome. To reconstruct dependencies within and between threads correctly, DotSim requires the identification of each instruction's linux thread id. Since gem5 had no mechanism to identify thread ids, the linux kernel was modified to write out a "thread_id" file identifying the relationship between thread id and cpu id over time. Similarly, generating traces from a full system simulator like gem5 brings an unwanted side effect; linux kernel code spills into the dynamic stream. Since linux kernel code is not representative of the ILP and TLP of the application being examined, particularly with the Alpha ISA where kernel code tends to be dominated by serialized PAL microcode, it is desirable to remove this code from the trace. Kernel

code has a distinct program code (PC) range addresses, thus, we filter the traces to exclude PC addresses in their range.

2.4.2.2 Trace Smoothing

The *TraceSmoothing* stage is responsible for two tasks: rewriting traces in terms of thread ids, and identifying synchronization constructs between threads and annotating them in the trace. As previously discussed, the *TraceGeneration* stage outputs traces in terms of cpu id, without distinguishing between threads. This stage rewrites the traces in terms of thread id using the "thread_id.txt" file which tracks the relationship between cpu id and thread id. In order to identify Pthread synchronization constructs, this stage calls a one-time script. This script disassembles the benchmark binaries and identifies the location of each Pthread function call. Identifying Pthread Function calls is made easier by compiling each benchmark with the "-g" flag for debug information inclusion, which labels all function calls. While in the general case identifying Pthread functions is automated, we found that for a subset of benchmarks (bodytrack, vips, x264, ferret, raytrace, facesim), the Pthread functions are used indirectly (*ie.* they are wrapped in other library functions or C++ classes). As a result, for these benchmarks the synchronization constructs were manually identified within the application binaries. Once identified, these synchronization construct function calls are stored in a "sync_construct.txt" file for use by the *TraceSmoothing* stage.

As the *TraceSmoothing* stage executes, the "thread_id" and "sync_construct" files are read and used to rewrite the trace to indicate the proper thread id and annotate the location of synchronization constructs. We note that the code associated with each synchronization construct function call code is replaced with a single DotSim syntax line that identifies what type of synchronization construct function

was used (*eg.* lock, barrier, condition variable, etc.) as well as what corresponding variables were used (lock id, barrier id, etc.).

2.4.2.3 DotSim Trace Language Conversion

The third stage converts the dynamic instruction traces into the DotSim trace language. This stage requires extensive knowledge of the ISA in order to identify all instruction types and dependency edges. Dependency edges can be control flow dependencies, register dependencies, memory dependencies and thread synchronization dependencies. All ISA instructions are stored in this stage, so they can be properly parsed and classified for dependency analysis as well as instruction classification. Unwanted instructions such as no-ops and cache hints, can be removed from the traces at this stage as well other instructions that do not involve registers or memory as operands. When identifying memory and store dependencies, this stage recognizes memory accesses across all threads, including load locks and store conditionals (in RISC ISAs) and standard load and store instructions.

2.4.2.4 Memory Footprint Optimization

DFG generation memory management can become problematic when creating a DFG via naive dependency analysis, since it is impossible to determine when a instruction will be dependent on a previous instruction. The simplest approach would be to hold all instructions which generate operands in memory inside the DFG generator as the DFG is processed. However, as the number of instructions goes to billions, this approach will quickly run out of memory. Fortunately, there are a limited number of registers; therefore, anytime an output register is rewritten, the previous instruction which wrote that register will no longer be needed. When this occurs the *Memory Footprint Optimization* stage inserts a delete node instruction into the trace, because it is impossible for any other instructions to dependent on it.

While this approach works well for registers, of which are typically limited in number, it does not work well for producer-consumer relationships through load and store instructions. Memory is not limited to just a few state elements but extends to a maximum of 2^{48} physical locations, making it impossible for the machine running the DFG generator to hold in its memory. Therefore, another mechanism must be found to delete operand producing store instructions when they are no longer used. Unfortunately, there is no way of knowing when a store is no longer needed in real time. As a result, the fourth stage is done after the first 3 stages have been completed and a trace has been stored to hard disk. This stage reads the trace in reverse order, "bottom-up", to identify when a store is no longer referenced by any further load instruction and insert delete node instruction. Figure 2.4 shows an example DotSim language segment where a load instruction last reads a given memory address. When parsing up from the bottom, when a memory address is first referenced by a load, that will be the last time it will be encountered in the forward direction. Therefore ensuring the store can be removed safely. Similarly, each time a store instruction to a given address is encountered in reverse order, the next load instruction to that address represents the last reference to that memory operand (prior to being overwritten by the next store). Thus after this load operation a delete can be inserted.

2.4.3 DFG Processing

Section 2.3 discusses DFG Generation at a high level, in this subsection we outline a few further details. After the preprocessing stages, a DotSim trace language file is generated and stored for further processing in the *DFG Processing* stage. Note, it is possible to skip the first four stages (the front-end) if one to were to write manually or use a script to generate DotSim traces. DotSim generates DFGs dynamically based on arbitrary resource constraints, or lack thereof. Traces only need to be

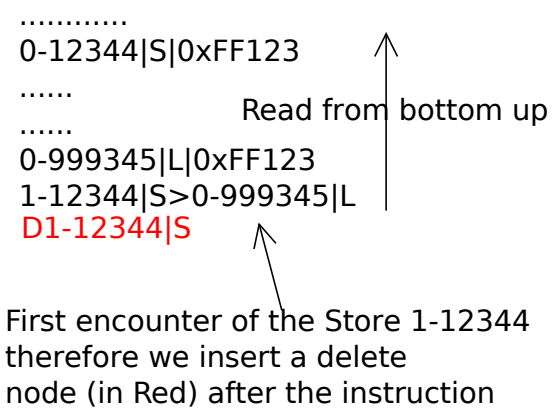


Figure 2.4: Memory footprint optimization using the bottom-up approach to locate the last touch to a given memory address.

generated once for each of the examined numbers of threads (N_{min}). Once the DFG is created, the height becomes the time in cycles to execute the program. As explained in Section 2.3, the critical path is also calculated. With the critical path, DotSim measures which threads are part of the CP as well as a breakdown of what type of instructions make up the CP. In the following section, we will discuss the micro-architectural features of DotSim, and why it is perfect for first order modeling of novel ideas.

2.5 Features

DotSim's main goal is to conduct limit studies on the execution of multi-threaded benchmarks in future processor architectures. DotSim has a varying degree of abstract modeling for microarchitecture behavior. In this section we enumerate the current features of DotSim as well as explain why DotSim is a useful tool for first order modeling of novel microarchitectural ideas. Currently DotSim supports both an abstracted Out-of-Order (OoO) and in-order machine. An initial goal was to understand the trade offs of TLP and ILP in multi-threaded applications. We leave it to

future work to implement additional constraints, to more accurately model current machines. In this section we enumerate and describe current available microarchitecture constraints that can be imposed in DotSim, then show why DotSim is a good simulator for implementing first order microarchitecture models.

2.5.1 Current Microarchitectural Features

1. *DotSim supports an abstract instruction window of arbitrary size for OoO execution with support for arbitrary issue widths.*

2. *DotSim supports modeling arbitrary, per-instruction-class execution times.*

3. *Multi-level cache modeling is supported with arbitrary shared and private levels. Users have the ability to change cache replacement policy, cache sizes, latency, and type of cache (direct, set- or fully-associative).*

4. *A synthetic Branch Prediction model based on arbitrary miss per thousand instructions (MPKI) rates is supported. This model can be easily extended to model a realistic branch predictor. As DotSim language provides enough information for a detailed branch predictor, although wrong-path instructions are not modeled.*

5. *Arbitrarily enabling or disabling thread synchronization semantics is supported. This will allow measuring limits of the upper bound of thread level speculation, as well as studying the trades off of ILP and TLP scaling.*

6. *DotSim provides a detailed analysis and statistics of the program's multi-threaded critical path (CP). It can output which threads are on the CP and the time spent per thread, as well as what type of instructions make up the cp.*

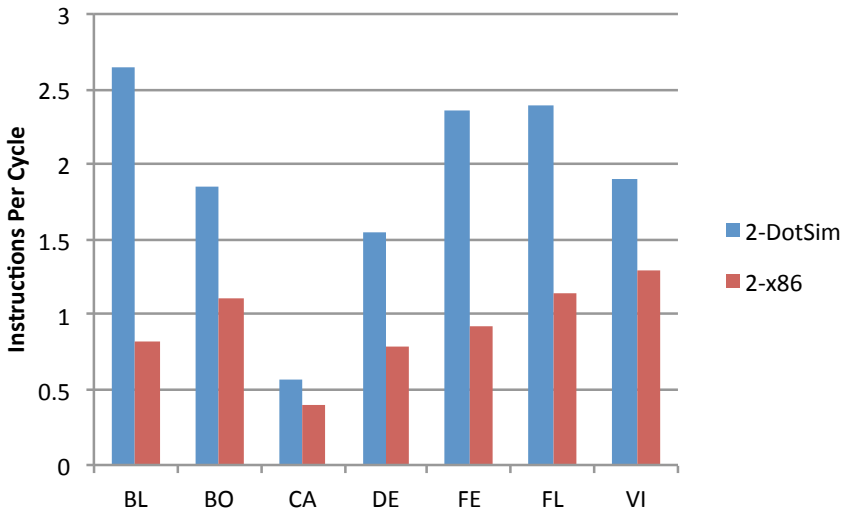


Figure 2.5: Comparing The ILP Limits Of An AMD Opteron 6167 Using DotSim

2.5.2 First Order Modeling On DotSim

DotSim models processor the microarchitecture in terms of it effects on the DFG of a program. Thus, one does not think of implementing hardware levels as one would in a typical execution driven simulator (or real hardware). Instead, users must consider how machine constraints effect the DFG. Therefore, the actual details of implementation are not required when implementing hardware level behavior. As a result of not requiring detailed layout of microarchitectural features, the complexity of implementing a novel idea in DotSim is greatly reduced. Thus, DotSim is ideal for first order modeling of novel microarchitectural approaches. In many cases first order modeling is sufficient enough to determine if an idea should be further investigated or dropped [33].

2.6 Case Study: The AMD Opteron 6167

In this section we perform a case study examining the ILP exploitable by a current microprocessor, an AMD Opteron 6176, versus the ILP bounds predicted by DotSim with a similar configuration. In this experiment, we configure DotSim to constrain cache, branch prediction, instruction window, and issue width to be that of the AMD Opteron 6176 processor, as well as respecting inter-thread synchronization construct semantics. We compare DotSim's predicted ILP limit to actual IPC results from the AMD Opteron 6176 (measured via the processor's built-in performance counters) to explore the closeness the ILP bounds.

We use the eight PARSEC benchmarks that our simulation infrastructure supports. To save space in the graph and text, the benchmark names are abbreviated to their first two letters. The abbreviations are as follows: **BL** - Blackscholes, **BO** - Bodytrack, **CA** - Canneal, **DE** - Dedup, **FE** - Ferret, **FL** - Fluidanimate, **VI** - Vips, and **X2** - X264. In each case the *sim-small* input set is used for both DotSim as well as on the real hardware.

For this study, we use the minimum number of parallel threads possible for each benchmark, N_{min}=2, meaning there will be at minimum 2 threads spawned for each benchmark [1]. We limited the thread count to minimize the noise introduced due to our simplified cache memory model (which models neither coherence latency nor interconnect delay). Here we focus on average ILP (ILP_{MT_AVG}), calculated according to Equation (3.2), as our primary figure of interest due to limited metrics available with the built-in AMD Opteron 6167 performance counters.

Figure 2.5 shows the resulting ILP_{MT_AVG} for each benchmark. In the figure we

[1]Note that, when configured as N_{min}=2, the PARSEC benchmarks will spawn a variable number of threads greater than or equal to that number. In particular, **BL**, **BO**, **CA**, and **FL** each spawn 2 threads, while **DE**, **FE**, **VI**, and **X2** spawn 12, 10, 4 and 6 threads respectively.

see that the ILP bounds and the measured IPC are closest for **CA**, this is unsurprising as the performance of **CA** is known to be severely restricted by cache size [4], which we model in our study. At the other end of the spectrum, **BL**, shows the widest gap between the ILP bounds and the actual exploited IPC. **BL** is known to be embarrassingly parallel. As a result its performance is most sensitive to functional unit latency and functional unit hardware hazards, which are not currently modeled in DotSim. Thus **BL** which consists predominantly of floating point computations, is throttled on real hardware with a limited number of floating point units.

Generally we see that DotSim models the maximum upper bound ILP to within 3x the measured IPC on the real machine. Thus the majority of ILP is restricted by the components that DotSim does model, (*e.g.* control flow, cache, issue width and instruction window) In contrast, using a perfect ideal machine with no constraints and removing inter-thread synchronization constructs yields, on average, 220x greater ILP versus the real AMD Opteron 6167. We speculate, that the remaining difference between the ILP bound and actual IPC measured is due DotSim not modeling cache coherency, wrong path execution modeling, reorder buffer size constraints, instruction latency, and functional unit latency and hardware hazards.

2.7 Limitations and Future Work

Though DotSim is quite flexible and capable of representing many microarchitectural features, it does have its limitations. DotSim's original goal was to identify ILP and TLP bottlenecks at the microarchitectural level, as a result it should be as simple as possible. In this section we list DotSim's limitations as a result of this design decision, as well as the most important features that should be implemented in future work.

2.7.1 Limitations

DotSim calculates the majority of its metrics after generating a DFG, therefore it is important that all dependencies are determined. This requires all operand producing instructions be held (specifically stores), until they no longer are dependent on. This can require intensive simulator memory requirements, depending on the benchmark and input size. So ideally a machine with a large amount of RAM is often required to run DotSim, depending on input size and benchmark. One way to alleviate this heavy memory requirement is to focus on the region of interest of each benchmark, thereby reducing trace sizes, thus reducing total memory requirements. Another issue is traces must be stored on a hard drive, so computation time can be bottlenecked by hard drive access time. This may be alleviated by using flash storage or running the trace files from a RAM drive. Further, as simulation speed can be limited by storage-system bandwidth, it is not recommended to run more than one simulation per hard drive. One way to alleviate the storage system sensitivity would be to skip the Memory Footprint Optimization stage and directly pass the traces via unix pipes directly to the DFG Processing stage. This approach, however, would mean that all the stages would have to be run for each configuration tested. Further, it might cause an even larger runtime memory footprint for the final DFG Processing stage, as last-touch stores would not be deleted.

2.7.2 Future Work

As it stands, the current release of DotSim fully supports Alpha ISA, utilizing a lightly hacked version of the gem5 simulator as a Trace Generation stage. DotSim preprocessing is thus currently limited to Alpha ISA, using gem5 [5] to generate traces. For future work we plan to implement support for x86 gem5 traces. This would require changes only to the preprocessing stages provided in the toolkit. We

currently have eight PARSEC Benchmarks, compiled for the Alpha ISA [15], executing without errors. As explained in Section 2.4, it is possible to use any benchmark, or other means of generating traces, as long as it is properly preprocessed to fit DotSim's trace language syntax. In future work we plan to expand the benchmarks supported to the full set of PARSEC 3.0 benchmarks on the x86 ISA.

Another important component of future work is to fully model an OoO engine, including implementing details such as a reorder buffer, load store queue, MSHR support and functional unit pipelines latencies. This would provide a significant increase in the cycle accuracy of DotSim, however it would come with substantial overheads in simulation time and require a slight revamp of the memory management system.

2.8 Summary

This chapter introduces DotSim, a trace-driven tool kit that is designed to explore the limits of instruction- and thread-level scaling and identify microarchitectural bottlenecks in multi-threaded applications. DotSim creates an instruction-level DFG from each thread in multi-threaded applications adjusting for inter-thread dependencies. The DFGs dynamically change depending on the microarchitectural constraints applied. In this paper, we show a case study that DotSim models the maximum upper bound ILP to within 3x the measured IPC on the real AMD Opteron 6176.

3. ILP AND TLP IN SHARED MEMORY APPLICATIONS: A LIMIT STUDY

With the breakdown of Dennard scaling, future processor designs will be at the mercy of power limits as Chip Multi-Processor (CMP) designs scale out to many-cores. It is critical, therefore, that future CMPs be optimally designed in terms of performance efficiency with respect to power. A characterization analysis of future workloads is imperative to ensure maximum returns of performance per Watt consumed. Hence, a detailed analysis of emerging workloads is necessary to understand their characteristics with respect to hardware in terms of power and performance tradeoffs. In this chapter, we conduct a limit study simultaneously analyzing the two dominant forms of parallelism exploited by modern computer architectures: Instruction Level Parallelism (ILP) and Thread Level Parallelism (TLP). This study gives insights into the upper bounds of performance that future architectures can achieve. Furthermore it identifies the bottlenecks of emerging workloads. To the best of our knowledge, our work is the first study that combines the two forms of parallelism into one study with modern applications. We evaluate the PARSEC multithreaded benchmark suite using a specialized trace-driven simulator. We make several contributions describing the high-level behavior of next-generation applications. For example, we show these applications contain up to a factor of 929X more ILP than what is currently being extracted from real machines. We then show the effects of breaking the application into increasing numbers of threads (exploiting TLP), instruction window size, realistic branch prediction, realistic memory latency, and thread dependencies on exploitable ILP. Our examination shows that theses benchmarks differed vastly from one another. As a result, we expect no single, homogeneous, micro-architecture will work optimally for all, arguing for reconfigurable,

heterogeneous designs.

3.1 Motivation

There have been many ILP limit studies to date[35, 44, 25, 16, 34, 2, 6]. These studies, however, often contradict each other, by making inconsistent assumptions with respect to ideal hardware capabilities, compiler capabilities and lacking consistency in the types of benchmarks used. In addition, these studies are almost exclusively more than 20 years old. Since then, the applications used in general purpose computing have evolved significantly to now work on much larger data sets with new and more complex algorithms. As such the applications in the prior studies are now largely outdated. Furthermore, we are aware of no study that has focused on the interaction between ILP and TLP. The purpose of our study is to understand the degree in which TLP extraction affects ILP availability. Understanding these trade-offs is important because exploiting TLP and ILP require different approaches to processor design. Exploiting TLP requires multiple-cores, while exploiting ILP requires larger cores. As it becomes more difficult to exploit the additional transistors gained by Moore's law, it is imperative to put these transistors to the best possible use with respect to performance and power efficiency. Analyzing the upper bound limits allows insight to the remaining parallelism. Understanding the trade-offs between ILP and TLP will help give insight on the optimal core counts and core makeup for a specific application with respect to power and performance.

3.2 Experimental Description

This section first discusses how our ILP and TLP limit study is conducted. We then go over the benefits of a trace-driven approach to study the limits of ILP in multi-threaded applications. Finally we cover our methodology for calculating the upper bound limits of benchmarks.

3.2.1 Limit Study

To measure the upper bound ILP limits found in shared-memory multi-threaded applications, we begin by assuming an ideal processor core model. This ideal model consists of infinite physical registers, perfect branch prediction, perfect memory-address aliasing (addresses are known, thus unrelated loads can move in front of stores). This model also includes unlimited hardware resources (unlimited instruction window, functional units, and single cycle memory latency). Furthermore, to explore the limits of TLP we remove all inter-thread synchronization constructs from the code (e.g. Locks, Barriers, etc.), while preserving true inter-thread data dependencies to ensure correctness. ILP and performance limits were then modeled via analysis of the application's true data dependencies (either through the register file or memory). From this starting point we then begin adding constraints, such as restricting window size and enforcing synchronization constructs. We then examine how the application's performance and ILP are affected by these constraints versus an ideal machine. Constraints are changed one by one or in combination of a few. This helps identify bottlenecks in the benchmarks under test. What sets our study apart from other studies is our ability to directly observe the tradeoffs in ILP and TLP in this class of parallel applications. As we increase/decrease the amount of TLP (by increasing/decreasing the number of threads) we are able to observe how this impacts ILP.

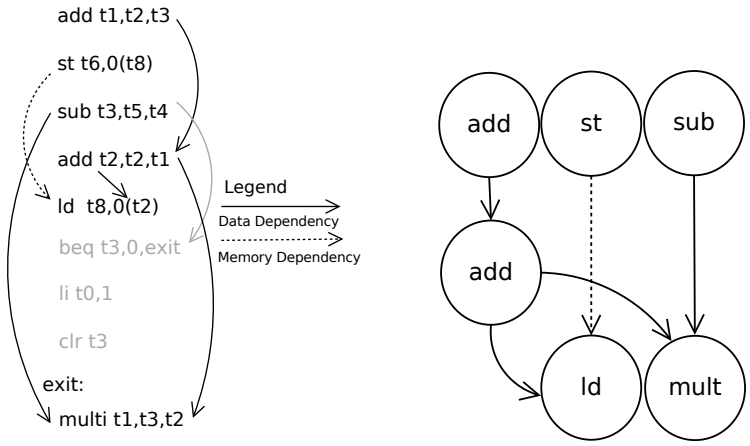
3.2.2 Trace-Driven Approach

Previous TLP studies were analytically performed with simple assumptions of application scaling [12, 19]. Analyzing benchmark traces that are generated from an execution-driven simulator (which simulates real machine behavior) can provide much more accurate bounds on performance. Our trace-driven approach allows us to

capture the actual dynamic instruction stream of multi-threaded benchmarks. These dynamic instruction streams have perfect branch prediction, memory-addresses are disambiguated, along with perfect memory (zero latency). A trace driven approach provides accuracy that an analytical approach cannot replicate. An analytical approach usually involves simplification to model real behavior. We chose to use execution traces rather than static code analysis since static code lacks memory disambiguation and control flow information.

Prior to feeding these traces into our trace-driven simulator, the traces are preprocessed once to work out all dependencies among instructions both intra-thread (through the register file and memory) and inter-thread (through memory). These traces are then fed into our trace-driven simulator, which then constructs a dynamic Data Flow Graph (DFG) for each thread, stitching together those threads with the dependencies between threads through memory. Figure 3.1 illustrates the process for a single thread. Figure 3.1a shows a simple assembly pseudo-code fragment with register data dependencies highlighted with solid lines, and memory data dependencies shown with dashed lines. In this example, the effective address of the store (st) and load (ld) alias to the same memory location forming a true data dependence through memory. Note that the branch instruction as well as the instructions on the branch not taken path are grayed out in the figure to represent the wrong path instructions. In our limit study we assume perfect branch prediction, therefore these instructions are not used in the DFG construction.

Figure 3.1b shows the corresponding DFG, that would typically be created with our simulator. The nodes represent instructions, and the edges represent producer-consumer data dependencies among the instructions. For the purpose of this limit study, all instructions are assumed to take one cycle. The maximum height of the DFG represents the number of cycles that this code fragment would take to execute



(a) Assembly fragment with dependencies highlighted. Wrong-path and control flow instructions grayed.

(b) Data-flow graph (DFG) for code in part (a).

Figure 3.1: Data-flow graph (DFG) for assembly fragment.

in an ideal machine. In the example, the height of the DFG is three, thus with infinite resources and one cycle per instruction the code would take three cycles to execute.

3.2.2.1 Calculating Single-Threaded Average ILP

Using the DFG it becomes easy to extract ILP of the program. Here we define ILP_{ST_AVG} of a given single thread (ST) to be the average (AVG) number of instructions that can be executed in each cycle under the given machine constraints. Equation(3.1), is used to calculate ILP_{ST_AVG}.

$$ILP_{ST\_AVG} = \frac{I_{all}}{H} \tag{3.1}$$

In this equation, I_{all} is the total number of instructions in the DFG, and H is the height of the DFG, representing cycle count of the ideal machine. For the DFG shown in Figure 3.1b, the ILP_{ST_AVG} is $\frac{6}{3} = 2$.

3.2.2.2 Calculating Multi-Threaded Average ILP

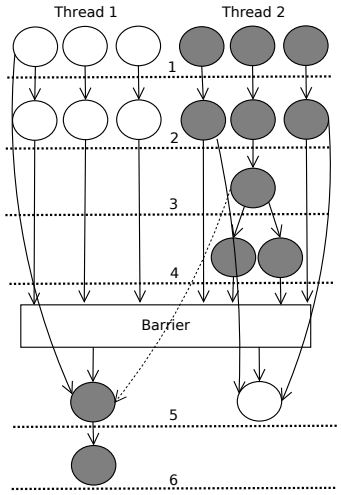
Extending this model to multiple threads in shared memory applications requires additional modeling of the dependencies between threads. Dependencies between threads are examined in two ways. First, to ensure correctness, the DFG generator models the true dependencies through memory caused by stores in one thread feeding to loads of the same address in another thread. Second, we also can model the dependencies in between threads caused by synchronization constructs inserted by the programmer in the code (e.g. locks, barriers, etc.). Figure 4.5b illustrates this process by showing a simplified parallel code fragment containing thread synchronization via a barrier. For this example, we assume two threads execute the "*worker_thread*" function simultaneously, each performing writes to parts of the shared array $x[]$. Later these threads read the array $x[]$, after being synchronized by the *pthreads_barrier_wait*() to ensure correctness. Figure 3.2b shows the DFG for these two threads with the thread synchronizing barrier in place. In this example, the height Thread 1's DFG (H_1) is six because it inherits Thread 2's height at the barrier (since it has the greater H of the two at this synchronization point). Here, H_2 is five. As per Equation(3.1), the ILP_{ST_AVG} of Thread 1 is 1.33 (The total number of instructions in Thread 1, I_1 is 8) and for Thread 2 is 2 (I_2=10). These results are averaged across threads to calculate the average ILP of multi-threaded (MT) application DFGs following Equation (3.2).

$$ILP_{MT\_AVG} = \frac{\sum_1^N (I_{alln}/H_n)}{N} \tag{3.2}$$

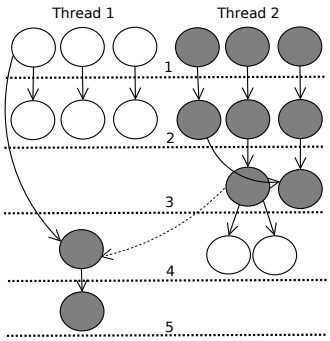
Here H_n, is the height of thread n's DFG, for threads 1 to N where N is the total number of threads in the benchmark. I_n is the total number of instructions in a given thread. For the DFG shown in Figure 3.2b, the ILP_{MT_AVG} is $\frac{1.33+2}{2} = 1.665$.

```
void worker_thread{
.....
...
while(loop)
.....
....
x[j] = y[j]*n;
pthread_barrier_wait (&barrier);
z[j]=x[i];
}
```

(a) Multi-Threaded pseudo-code fragment.



(b) DFG for code in part (a) assuming two concurrent threads. Both in-memory dependencies and thread synchronization constructs honored.



(c) DFG for code in part (a) with thread synchronization constructs removed.

Figure 3.2: Multi-threaded code fragment and associated DFG, with and without thread synchronization constructs honored. Darkened nodes represent instructions along the critical path.

3.2.2.3 Calculating Multi-Threaded Critical Path ILP

ILP_{MT_AVG} calculates the program's ILP under the assumption that the application scales ideally with thread count. Unfortunately applications do not typically scale perfectly, hence we introduce a new ILP metric in this section which gives attempts to provide a more useful estimation of the effective ILP for multi-threaded applications. Figure 3.2 illustrates the issue. In Figure 3.2b, although H_1 (the H of Thread 1) sets the lower-bound limit on execution time for this example, we note that Thread 1 alone does not determine the overall height (H_{max}) of this multi-threaded program. Here, speeding up execution in Thread 1 would not lead to a significant performance increase. Because, although Thread 1 has a higher H after the barrier, it inherits the H of four from Thread 2 at the barrier. Thus both Thread 1 and Thread 2 have an impact on the overall H_{max} of the program. Instructions from Thread 2 prior to the barrier and instructions from Thread 1 after the barrier form the *critical path* (CP) of the application. We define the program's CP (indicated by the darkened circles in Figure 4.5b) as the dependency chain of *thread segments* through the program that determines the H_{max}, *ie.* the execution time of the program in the limit. Much insight can be extracted from per-thread ILP as well as the critical path ILP. In this example, the number of instructions on the CP is 11, and the CP takes six cycles to execute, therefore we define the ILP_{MT_CP}=1.83 instructions/cycle for this code. To calculate ILP_{MT_CP}, equation(3.3) is used.

$$ILP_{MT\_CP} = \frac{\sum_1^k I_{kn}}{H_{max}} \tag{3.3}$$

In this equation, I_{kn} is the segment of instructions that are under the height (H_{max}) defining segments of each thread in the DFG.

Further insight can be gained on the limits of ILP and TLP scaling when thread

synchronization semantics are removed. This approach allows an exploration of limits of cross-synchronization boundary speculation. Figure 3.2c shows the DFG for the code fragment when all thread synchronization semantics are removed. In this case, correctness is ensured by continuing to enforce direct producer/consumer relationships between threads through memory. As the figure shows the overall H_{max} is reduced to five because two, now non-critical, instructions in Thread 2 are removed from the program's CP. The new CP has I_{CP}=10 and an ILP_{CP}=2 due to the increased efficiency of the resultant code path. We denote this measure of the ILP of the critical path without synchronization semantics as $ILP_{MT\_CP\_NS}$. Note that here we are again measuring the dependency chain of *thread segments*, not solely the dependency chain of instructions themselves. Thus, there are two instructions at $H = 3$ in Thread 2 that are counted as critical, despite the fact that one of them is not directly on the critical path connected to Thread 1.

Defining a CP metric, is a difficult and somewhat fraught question, with several possible derivations. Ultimately, we chose the current CP metric because we felt it provided more insight than the alternative approaches. In particular, the current metric captures the average dynamic ILP width of the CP segment in question, thus indicating how wide a core must be to achieve the performance shown. Alternate approaches would not show the true ILP width of the CP and thus provides less useful information about the desired width of the machine needed to execute it.

3.2.2.4 Threading Inefficiency

While ILP_{MT_CP} provides information about the width of the machine needed to achieve a given performance on multi-threaded applications, it does not give a full picture of the relative balance between threads. Here we introduce a new term, called *threading inefficiency* (TI). TI is a measure of the relative imbalance between

the CP (in terms of instruction count) versus the average thread. TI is calculated according to Equation (3.4).

$$TI = \frac{I_{CP}}{\frac{I_{all}}{N}} \tag{3.4}$$

In this formula, I_{CP} is the total number of instructions in the CP and I_{all} is the total number of instructions in all threads.

Analyzing the TI, particularly as thread count increases, provides insight into the overheads of scaling out to many cores. The CP is the longest path in the execution of a program, therefore the maximum speed up of a program is limited by its CP. When TI is greater than 1, the CP is greater than the average thread length, it indicates the workload is uneven. For example, a TI of 10 means the CP is 10 times larger than the average thread length, which we would interpret as the application being highly imbalanced.

2

3.3 Evaluation

In this section we first discuss some details of the methodology of our study and then present our results.

3.3.1 Methodology

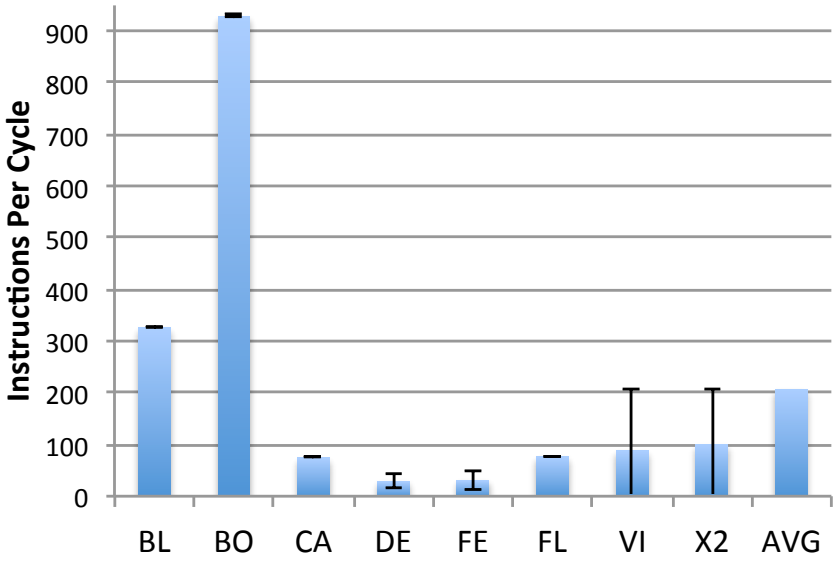
To conduct the work presented here, we generated dynamic instruction stream traces using the gem5 Simulator [5] and the PARSEC Benchmark Suite [4] compiled for the Alpha ISA [15]. All the PARSEC benchmarks that our simulation infrastructure supports are presented. To save space in the graphs, the benchmark names are abbreviated to their first two letters. The abbreviations are as follows: **BL** - Blackscholes, **BO** - Bodytrack, **CA** - Canneal, **DE** - Dedup, **FE** - Ferret, **FL** -

Fluidanimate, **VI** - Vips, and **X2** - X264.

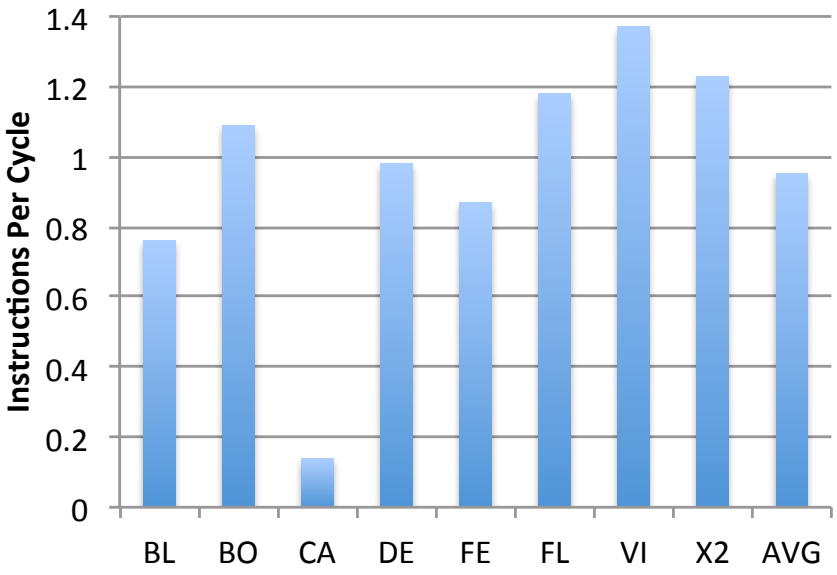
Once traces were generated, we pre-process them by identifying data, instruction, and thread dependencies, similar to the example given above Figure 3.2. An individual trace set was generated for each of the examined numbers of threads (N_{min}). These traces are then fed into our off-line analysis tools which create a dynamic DFG. For this chapter our tools dynamically generate different DFGs dependent upon window size and the presence or absence of thread synchronization semantics, as shown in Figure 3.2.

One issue with a trace-driven approach, were traces are generated once and then analyzed off-line, is lock acquisition order may change under different execution constraints, particularly in benchmarks which utilize fine grained locking. In the set of PARSEC benchmarks which we are able to execute in this infrastructure, this only applies to Canneal. In our prior work we found this effect to cause a relatively minor impact in the measured performance, generally <10%.

Due to time constraints, small input sizes are used for all figures. Only instructions in the Region of Interest (ROI) were examined. In order to ensure that the integrity of this idealized limit study was not effected by input size we selected a few benchmarks to run using small, medium and large input sizes. After studying the results closely we found that the behavioral trends extracted from small input were mirrored in the medium and large inputs as well. We also ran the native input on real machines, and found the trends remained the same.



(a) ILP_{MT_AVG} for N=2



(b) ILP on a real machine (2.3 GHz AMD processors)

Figure 3.3: ILP Limits

3.3.2 ILP and TLP Limit Study Results

Our limit study explores seven important questions as described in Section 3. We answer these questions by evaluating PARSEC benchmarks in our limit study. We believe answering and analyzing the following questions will help provide insight into what future architectures should look like for next-generation workloads.

3.3.2.1 What is the Upper Bound on ILP?

Determining the upper bound on ILP available in multi-threaded benchmarks helps quantify its availability for exploitation. If there is a plethora of ILP being left on the table, then this will be clear motivation to develop better techniques to exploit ILP. To answer this question we simulated our traces with unlimited resources and no thread synchronization semantics however data and register dependencies are still preserved. Although these benchmarks do have single threaded versions, we found that they behaved significantly different from the multi-threaded versions. There are many algorithmic transformations that are tied to the sequential versus the parallel versions of the code. We therefore felt the serial versions were not similar enough to provide useful results. To have a consistent baseline, we start at the minimum number of parallel threads possible for each benchmark, N_{min}=2, meaning there will be at minimum 2 threads spawned for each benchmark. [1] We calculate the upper bound ILP (ILP_{MT_AVG}) using the equation(3.2).

Figure 3.3a shows the resulting ILP_{MT_AVG} for each benchmark. Here the whiskers show the standard deviation (SD) of ILP among threads. The ILP_{MT_AVG} varies from 29-929 Instructions/Cycle with an average across the benchmarks of 200. As shown by the whiskers, **FE**, **DE** , **VI** and **X2** show a noticeable variance in the

[1]Note that, when configured as N_{min}=2, the PARSEC benchmarks will spawn a variable number of threads greater than or equal to that number. In particular, **BL**, **BO**, **CA**, and **FL** each spawn 2 threads, while **DE**, **FE**, **VI**, and **X2** spawn 12, 10, 4 and 6 threads respectively.

ILP difference among threads, while the other benchmarks show little variance.

The **first observation** we derive from this data is that there is significant variance in ILP between benchmarks. For **DE** and **BO** (lowest and highest respectively), the ILP varied by a factor of 32x. **DE** finds and removes redundancies from data streams with a technique called deduplication. The application has heavy communication among threads. Despite having several parallel stages, each stage is highly dependent on the previous stage thus restricting ILP. **BO** processes images/videos and keeps track of a human body. Its has high ILP because the frame input is fixed and all processing is dependent on that frame. The input does not alter, and therefore, can be parallelized very effectively (the highest of all benchmarks).

For comparison, we ran these benchmarks on real machines, with N_{min}=2, using the native input set. We then measured the ILP_{MT_AVG}; results are shown on Figure 3.3b. This yielded our **second observation**, not only do some of these benchmarks fare poorly (**CA**) in real machines in terms of ILP, they are orders of magnitudes less than what could theoretically be achieved. There is no doubt, a great deal of ILP is left on the table in these applications. This argues that greater effort should be spent in finding ways to exploit ILP in next-generation hardware. These results are motivation to push for more research in aggressive speculation to maximize ILP gains.

3.3.2.2 What is the Threading Inefficiency of Each Benchmark?

Figure 3.4 shows the TI, as defined by Equation 3.4, for all benchmarks from N_{min}= 2-64. For the case of **X2** and **VI** , we were unable to generate values of N_{min} greater than 2 due to infrastructure problems, so they will be left out of the analysis in this and the following questions. This experiment uses the same configuration as discussed in Section 3.3.2.1. From Figure 3.4 we notice two distinct groups of

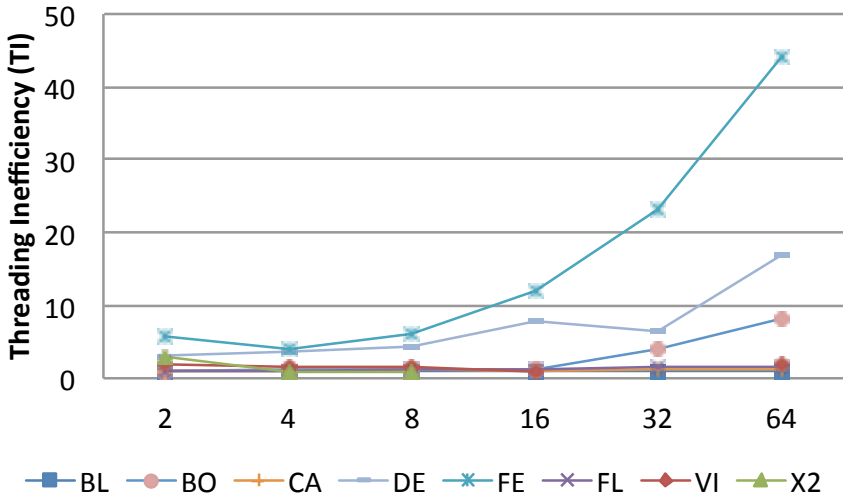


Figure 3.4: Threading inefficiency (TI) of PARSEC benchmarks for $N_{min} = 2$ through 64.

behaviors. In the first group, containing **CA**, **BL**, and **FL** , TI increases slowly with thread count and TI never exceeds 2x. **BL** shows the best TI, approximately 1 for all N_{min}. **BL** represents an embarrassingly parallel program, as it remains well balanced as threads increase. In **BL**, the problem space is evenly partitioned among the cores and there is very little interaction among the threads until the program completes [4].

The second group consists of **FE** , **DE** , and **BO** . In these benchmarks TI increases dramatically with thread count. The underlying parallel algorithms in this group involve a great deal of inter-thread dependencies at a high level. This increase still occurs despite removing all thread synchronization overhead (inter-thread data dependencies are always respected to ensure correctness). Bienia et. al. found these benchmarks to be the only ones using the Pthread condition synchronization function in our study [4]. Pthread condition variables are used by a thread to suspend or wake up other threads for the purpose of synchronizing data. These results indicate

that, despite ignoring the condition thread synchronization semantics, overheads of coordinating parallelization remain high for these applications. The parallel threads generated by this group also spawned various types of worker threads that had different types of jobs to do. These threads relied on prior stages of thread pools to reach a certain checkpoint before executing. This leads to our **third observation**, there is a dramatic increase in TI for benchmarks that require threads to heavily depend on each other (a good rule of thumb is when a parallel program uses a significant number of Pthread condition instructions). In other words the more complex the parallel algorithm is, the more difficult it is to scale. While this observation may be thought of as "intuitive", what we add is the quantification of scaling overhead. Furthermore, we find that depending on the complexity of the benchmark, the penalty of scaling can be amplified. It should be emphasized that the degree of inefficiency varies among benchmark. **FE** had the worst TI (44), and **BL** had the best (1). Knowing the TI of each benchmark is not enough to to optimize your returns on hardware, rather it is a combination of resources available with respect to CP, ILP, and cycle time. Threading inefficiency is a useful metric, which we will use to help interpret data we present in the questions below.

3.3.2.3 What is the Impact on ILP as Core Count Scales?

In this experiment we study the effects of the CP's ILP (ILP_{MT_CP}), as we increase TLP (by adding more threads). Here we ran benchmarks setting N_{min} to 2 through 64, simulating ideal cores. We again ignored all thread synchronization semantics, thus following the discussion for extracting $ILP_{MT\_CP\_NS}$ described at the end of Section 3.2.2.3 , and illustrated in Figure 3.2c. Figure 3.5 shows the ILP of the CP for the varying N_{min}. Showing the ILP of the CP gives a much more accurate understanding of the tradeoffs between ILP and TLP caused by inefficiencies in multi-

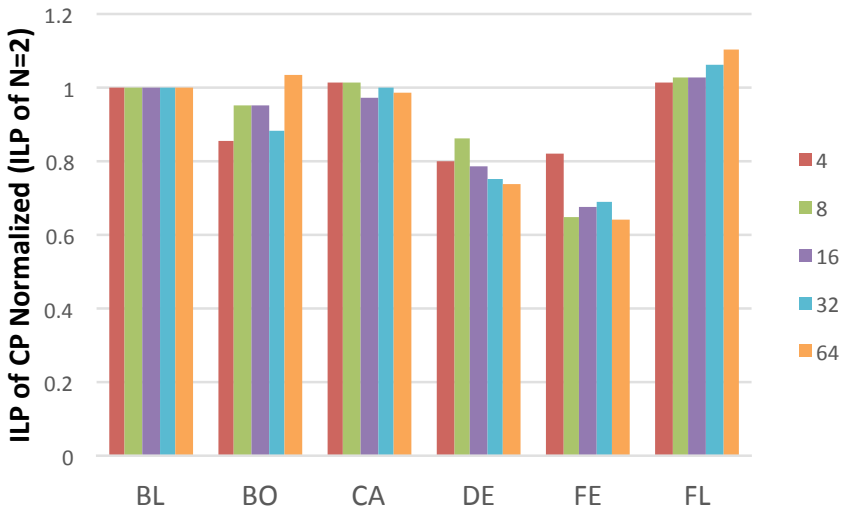


Figure 3.5: Normalized ILP of the CP without synchronization semantics, N_{min}=4 through 64 ($ILP_{MT\_CP\_NS}$). Normalized against CP ILP for N_{min}=2.

threaded load balancing. The CP takes into account all threads of a program. It is typical for a benchmark to spawn additional threads as it continues to execute. For some benchmarks such as **FE**, setting N_{min}=64 will yield a CP which touches on as many as 258 threads (many of these threads are short-lived). It is important to understand that an increase in ILP does not necessarily indicate it is better for the overall program in terms of performance and speedup. One must take into account the CP's instruction count and the application's TI to understand the presented CP ILP values.

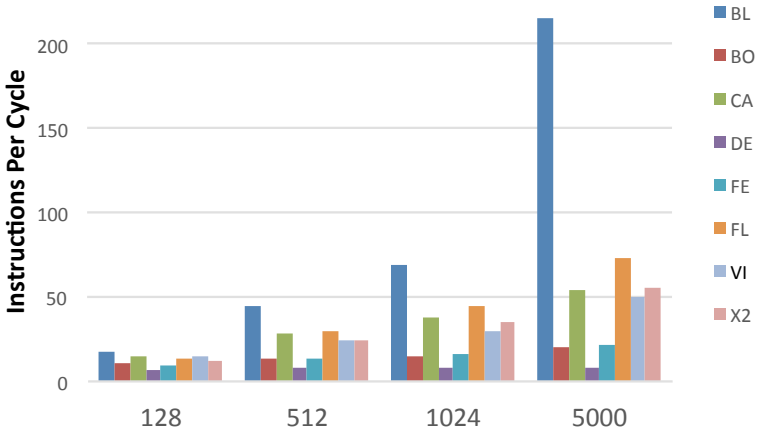
From Figure 3.5 we see **CA**, **FL** and **BL** are the only benchmarks that only show a small loss in ILP as N_{min} increases. Furthermore, in **FL**, the ILP actually increases forN_{min}=32 and 64. It is worth noting that these are the simplest of all benchmarks, in terms of parallelism model (they are data-parallel). For these benchmarks it would make sense to keep core sizes the same as cores are added to compensate for the increase in TLP. **DE** and **FE** see ILP reductions of 27% and 45% respectively

as you scale out to 64 cores. For these applications, a good tradeoff that conserves power as N_{min} increases is to reduce the size of the cores. **BO** is the most interesting as the CP ILP is best at 2 and 64, it shows less ILP for intermediate N_{min}. Thus the optimal ILP for **BO** is found at 64 cores. For this benchmark, it makes sense to maximize the size of all cores for N_{min}=64 cores, however, note that this benchmark has among the worst TI (Figure 3.4). This brings us to our **fourth observation**, as you increase the number of cores, the optimal core size to optimize performance with respect to power varies greatly with application (it should match ILP trends). Creating extra overhead due to scaling often gives additional ILP, but that does not speak to whether it is beneficial or not, in order to properly process this, the TI must be taken into account (shown in Figure 3.4).

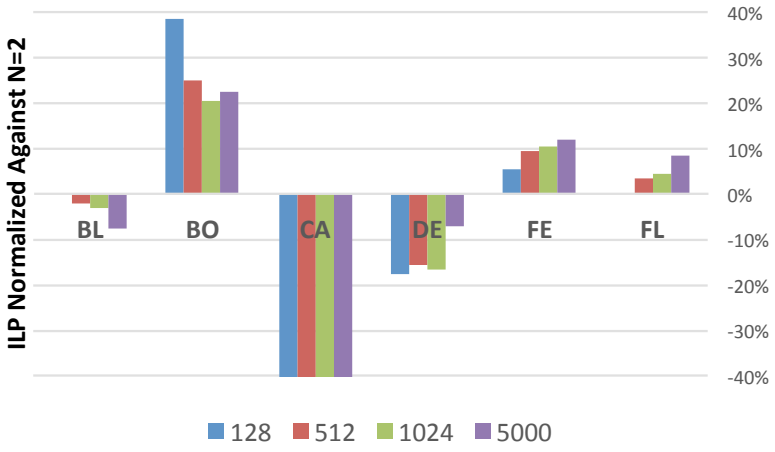
3.3.2.4 What is the Impact of Instruction Window Size Restrictions?

In this experiment, we vary the simulated instruction window size. This experiment explores the search distance required to achieve significant ILP gains. The results should also help inform design decisions with respect to die area and power. Figure 3.6a shows the $ILP_{MT\_CP\_NS}$ with window size constraints placed on each benchmark, for N_{min}=2. From the results, it is clear that window size heavily impacts ILP. As the figure shows, an instruction window size of 128 restricts max ILP to roughly 7-18 depending on the benchmark. Compared to the results for an infinite window (Figure 3.3a) this reflects a loss in ILP available of 4-59x. When using a 5000 window size the applications gain more than half the ILP that we see in an infinite window for **CA**, **BL** and **FL**.

Looking at instruction window sizes of 128 and 512 (a 4x increase in size), we see slightly above 2x the returns in ILP for **BL**, **CA** and **FL**. As noted previously these are the simplest benchmarks in terms of parallelism model (all 3 are using



(a) $ILP_{MT\_CP\_NS}$ for N=2



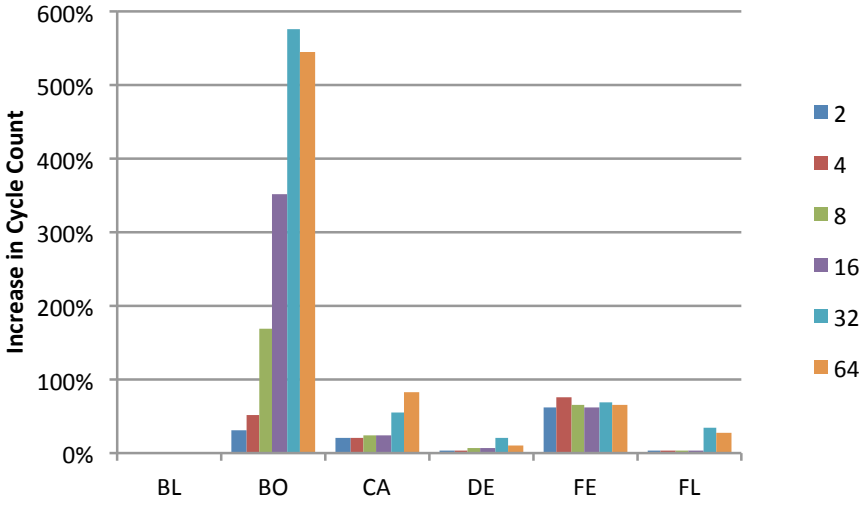
(b) $ILP_{MT\_CP\_NS}$ for N=64, normalized against N=2

Figure 3.6: Instruction window constraint impact on $ILP_{MT\_CP\_NS}$.

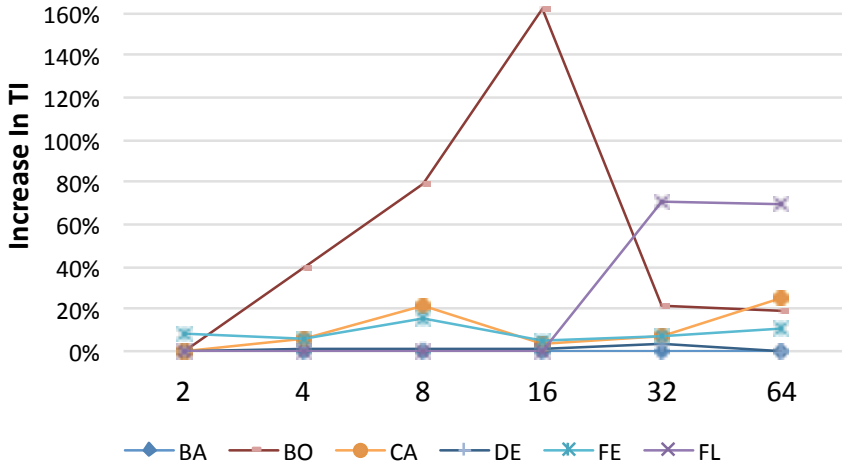
data-parallel algorithms [4]). Diminishing returns set in for ILP as window sizes increase from 512 to 1024 and finally to 5000. For the rest of the benchmarks the bulk of the ILP is found beyond instruction windows of size 5000. These benchmarks are much more complex, and as a result this additional ILP is likely from different phases of the parallel algorithm. They also have different parallel stages, which have significant instruction parallelism relative to each other. Our **fifth observation** is that we see that ILP is being heavily restricted by window size.

In half the benchmarks, most of the ILP can be mined from within a distance of 5000 instructions (**BL, CA, FL**), while in the other half (**X2, VI, FE, DE, BO**), the ILP is found much further than a distance of 5000 instructions away, making it more difficult to capture using a traditional instruction window. Improving performance in the first group might involve increasing the instruction window, while the second group would require much more aggressive speculation techniques. We speculate near ILP (<5000 windows) is likely to include what is traditionally considered to be Data-level Parallelism (loop bodies etc.) while far ILP is more likely to come from different program phases and related phenomena ILP as mentioned prior.

In Figure 3.6b we show the change in $ILP_{MT\_CP\_NS}$ comparing N_{min} of 2 and 64. We find the ILP of the CP changes as you increase N_{min}. In some benchmarks the increase in TLP takes away from the ILP when imposing window constraints. For **BL**, it makes little difference, while for **FE**, **FL** and **BO** we notice an increase in ILP as we increase TLP. However for **CA** and **DE** we notice ILP is reduced. Therefore depending on the benchmark, increasing TLP does not necessarily decrease the amount of ILP when using window size constraints. When we do observe an increase in ILP it is worth noting that this at a cost of an increase in threading inefficiency.



(a) Impact of thread synchronization semantics on H.



(b) Impact of thread synchronization semantics on TI.

Figure 3.7: Impact of thread synchronization semantics.

3.3.2.5 How do Thread Synchronization Semantics Effect ILP?

Thread synchronization semantics (i.e. locks, barriers and condition variables) are inserted by programmers to synchronize data access between threads, with the goal of removing races and ensuring correctness. Often theses semantics are implemented very conservatively, sacrificing program performance to reduce programming time and complexity. For example, a programmer may insert a lock to synchronize all accesses to an array even when different threads are not actually working on the same elements of that array, and hence there is no actual data dependence between threads accessing the array. Furthermore, there are often independent instructions beyond a synchronization semantic that could be executed in parallel while waiting on the synchronization semantic. In this section, we explore the potential benefit of speculation beyond synchronization semantics on the ILP of the CP. Thus, here we move from measuring $ILP_{MT\_CP\_NS}$ to ILP_{MT_CP} as illustrated in Figure 3.2b.

Intuitively, we expect that removing the thread synchronization semantics imposed by the programmer will reduce H, the DFG height (ie. the estimated cycle count). Here we quantify the actual impact on performance by focusing on the changes in H of each benchmark. When simulating without synchronization primitives, we still honor the true data dependencies between threads, therefore applications behave as if an idealized, fine-grain synchronization was used. For this experiment, we compared H for each benchmark at each N_{min} with thread synchronization semantics enabled against a parallel run with thread synchronization semantics ignored (as was done in all experiments to this point in the chapter). A large increase in H with thread synchronization semantics enabled would provide motivation for revisiting TLP exploitation techniques as well as motivation for creating new techniques to speculate past thread synchronization semantics as discussed

in Section 3.4.3.

Figure 3.7a shows the increase in H with thread synchronization semantics enabled versus with thread synchronization semantics disabled. Figure 3.7b shows increase in TI due to thread synchronization semantics. **BL**, being an embarrassingly parallel program, shows no effect from thread synchronization semantics. **CA** is an interesting case, as it contains the finest-grain synchronization of all the applications, using load locks and store conditional instructions directly to create atomic locks. It also uses high level barriers to synchronize all the threads. **CA** at 64 threads, shows an 80% slow down when enabling thread synchronization semantics. In this case it is the use of these barriers that inhibits performance. **FE** has the biggest increase in cycle time at $N_{min} = 4$ (slowing down nearly as much at 80%), which is interesting since it is a pipelined parallel algorithm (multiple stages in the program applications, where some stages are parallel), spawning many more than the minimum N threads for each case. **FL** does not see any performance impact from thread synchronization semantics until N_{min}=32-64. **DE** , another pipeline parallel benchmark, has very minimal slowdown when enabling thread synchronization semantics, along with **BL** it stands to benefit the least. **BO** is particularly interesting as it has an significant impact on cycle time due to thread synchronization semantics. This leads to our **sixth observation**, there is sufficient motivation to develop new and aggressive thread dependency speculation techniques, particularly for $N_{min} \geq 32$. Its worth noting, although intuitive, almost every benchmark experiences a significant increase in threading inefficiency due to synchronization semantics.

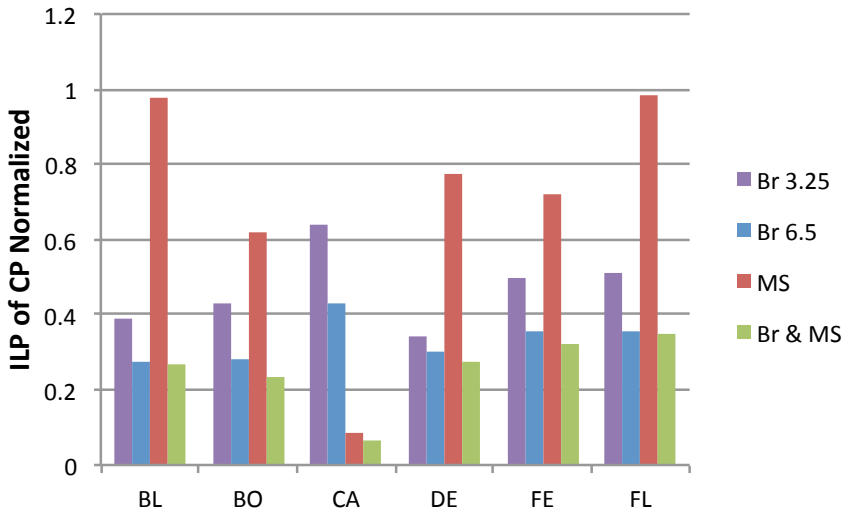
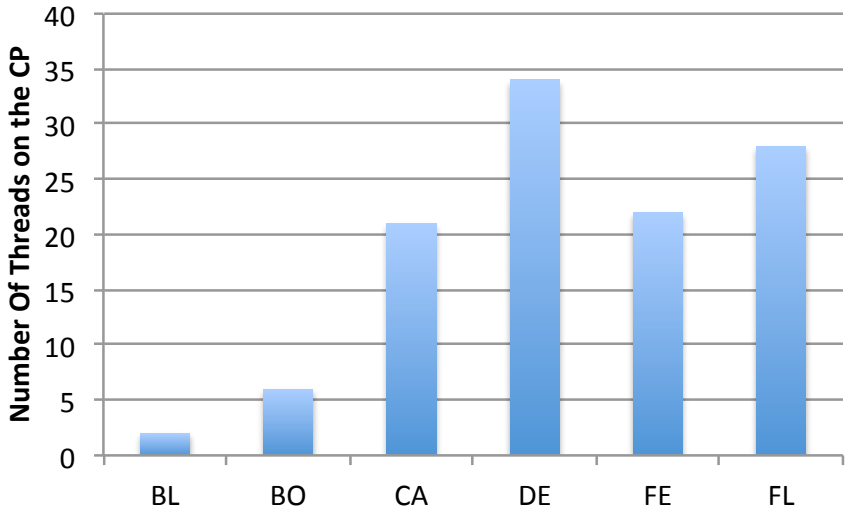


Figure 3.8: ILP_{MT_CP} with Branch, Memory Latency, and both, normalized against Nmin=64, thread semantics enabled and a 128 instruction window.

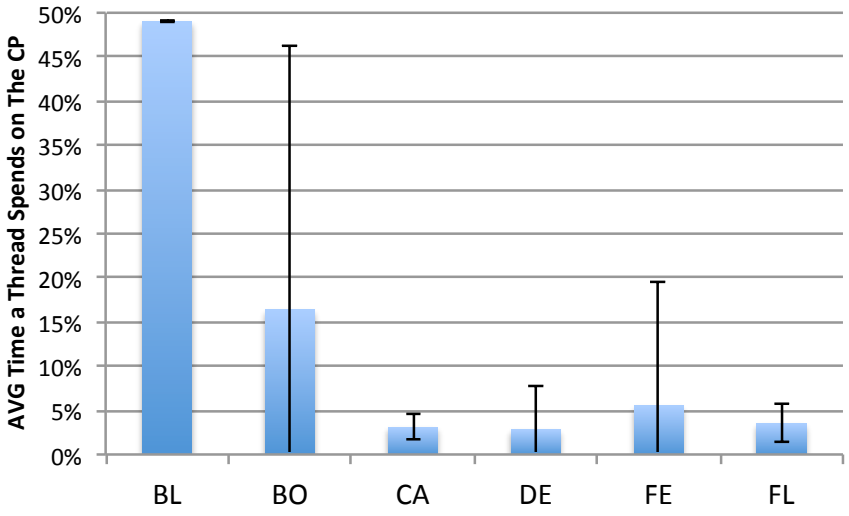
3.3.2.6 What is the Impact on ILP of Realistic Memory System Latency and Branch Prediction Accuracy?

To explore the impact of realistic constraints on memory system latency and branch prediction accuracy we perform four experiments. In the first two experiments we add realistic branch prediction (Br) modeling different prediction accuracies. In the third experiment we add realistic memory latency using a cache simulator (MS). Finally, for the last experiment we add both MS and Br simultaneously. For these experiments we set N_{min}=64, with thread dependencies enabled, using a 128 instruction window. For branch prediction simulations we implement a realistic model of 6.5 branch misses per thousand instructions (MPKI), which represents current typical branch predictor performance [22]. we add realistic memory latency using a cache simulator (MS). For our realistic memory system we implemented simple cache hierarchy, modeling private L1 and L2 caches and a shared L3 last-level cache. The

sizes of each cache are 256KB, 512KB, and 2MBs respectively. The three levels of the cache have an access delay of 1, 10, 30 respectively, a L3 cache miss results in a 150 cycle delay. Here we chose 2MB of L3 cache since we are using the small input set of PARSEC the memory footprint is likely smaller than native input sets. However, the trends should remain the same for larger input sets when using a larger last level cache. Figure 3.8 shows our results for these experiments. The results show that there is an average 31% reduction in ILP_{MT_CP} when adding memory system model delay, and an average 67% reduction when adding a realistic branch misprediction rate of 6.5 MPKI (Br 6.5). **BL** and **FL** have very little ILP_{MT_CP} degradation when implementing a realistic memory system. Interestingly, **CA** has the worst cache ILP_{MT_CP} performance, being reduced by 92%. The results for **CA**, are consistent with Bienia et .al's [4] work, where **CA** is known to be severely restricted by cache size. Generally, with the exception of **CA**, we find that branch misprediction has a nearly uniform, and drastic impact on the ILP available. To further explore this impact, we improved the MPKI by a factor of 2, (from 6.5 to 3.25) shown as Br 3.25 in the figure. Improving MPKI by a factor of 2 results in an ILP improvement of 14%-51% depending on the application. Our **seventh observation**, is branch prediction is the stronger bottleneck of the two under these constraints. These results provide motivation to continue expending resources to improve branch prediction, despite it being a mature field of research [21, 22, 40]. In our last experiment, we add both Br (6.5 MPKI) and MS constraints simultaneously. The results in Figure 3.8 show the combination tends to follow the most constrained of either Br or MS depending on the benchmark.



(a) Number of threads on the CP.



(b) Average time spent on CP. Whiskers show one standard deviation.

Figure 3.9: Characteristics of the critical path (CP).

3.3.2.7 What are the Defining Characteristics of the Critical Path?

Finally, we examine the characteristics of the critical path (CP) for each benchmark. In this experiment, we utilize a configuration that most realistically resembles a forward looking chip-multiprocessor, 64 cores, with a 128 instruction window using a realistic memory latency and branch prediction (6.5 MPKI), as well as thread synchronization semantics enabled. We recorded what threads were on the critical path, as well as how much of the CP was made up of each thread. We then filtered the results and removed any thread on the CP that made up less than 1/(Total Threads on the CP) when $N_{min} = 64$ (some threads spawn more than others as explained in Section 3.3.2.1). Since the CP is the longest path from start to end of execution, it makes sense to speed up the CP by using the highest performing cores available. By characterizing the CP, we can determine the practicality and or difficulty of CP thread migration for the sake of acceleration. Figure 3.9 shows this data along two axes.

Figure 3.9a shows the number of threads that make up the CP of each benchmark. The bars in Figure 3.9b show the average time a thread spends on the CP, while the whiskers on each bar show one SD of thread time on the CP. Using these two graphs, we are able to highlight a few observations. **BL** has two threads on the CP, each containing 50% of the CP. **FE** had 34 threads on the CP that we considered critical; interestingly, it spawned a total of 258 threads. Out of the 34 critical threads in **FE** one thread dominated the CP by occupying 62% of the time. In **BO**, there were 6 threads on the CP, with one thread dominating, occupying 77% of the time. Thus we feel it would be beneficial to have one or two bigger cores, to execute the dominant threads on the CP for both benchmarks. The other benchmarks had many threads making up their CP. For these applications, it is likely still advantageous to have a

few larger cores to execute the CP and migrate threads to those cores on the CP when they become critical, this is particularly true for applications with poor TI. This follows into our **eight observation**, there exists a potential for performance gain by creating larger cores and migrating threads to that core when they are on the CP. This approach, however, will require research to explore techniques that can identify thread criticality in real time [11, 42].

3.3.2.8 Summary Of Observations

In this section, we explore the limits and trade-offs between ILP and TLP in modern, shared-memory multi-threaded applications. Here, we summarize our observations from the experiments conducted. Current architecture designs are far from achieving even a fraction of the ILP that is available in these applications. Clearly, there remains a huge headroom for improvement with respect to ILP exploitation. This suggests for more creative techniques to maximize ILP while maintaining power requirements. Although traditionally ILP extraction has been viewed as extremely power intensive, recent commercial processor designs indicate that this may be possible. In particular, comparing Intel's Haswell versus Ivy Bridge processors, through a concerted effort to reduce power, Intel has managed to reduce power consumption by 50% while slightly increasing ILP extraction for two processors *in the same 22nm process technology* [1]. Even within a distance of 128 instructions, there remains a substantial amount of ILP that is not being mined by current architectures. Within a 5000 instruction window the simplest data-parallel programs can extract nearly half the upper bound of ILP calculated in this chapter. In more complex benchmarks, such as those that use parallel-pipeline algorithms, a majority of the ILP is much further away and likely impossible to capture using a traditional instruction window. Therefore there is a good motivation to *increase window size*, as well as to

find innovative ways to capture ILP that is much further away using a much more *aggressive speculation techniques*. In order to explore the impact of realistic design constraints on ILP we added simple branch prediction and memory system models, and found that branch prediction is more important of the two with respect to ILP under these constraints. Despite branch prediction cost overhead and the maturity of this field, *it may be worthwhile to expend resources to improve branch predictors, potentially even over increasing cache size* [21, 22, 40].

We find that increasing TLP does effect ILP, but the trade-offs are not consistent among all benchmarks. The trade-offs depends on the benchmark, a good rule of thumb for parallel algorithms: the simpler the parallel algorithm the less likely TLP will negatively effect ILP. Therefore as core counts increase, core size *should be dependent upon the application*. From a practical viewpoint, this argues for heterogeneous architectures (only powering up core size appropriate for the given application) or dynamic architectures. We find that thread synchronization semantics can greatly impact program performance, although the amount depends on the application. There is as much as a 6x slowdown due to these semantics. The overhead of thread synchronization semantics on performance become quite noticeable as programs scale to larger numbers of cores. Therefore, there should be more *effort in trying to speculate beyond synchronization semantics*.

As cores continue to scale the more likely there will be an increase in load imbalance. This increased load imbalance puts pressure on certain threads causing them to become more critical than others. This argues for heterogeneous designs which map threads to high performance cores when those threads become critical. Since we find the critical path often migrates from thread to thread many times during the application's runtime, we find that there is a critical need to *identify critical threads during runtime*, to enable performance critical thread migration. Generally in our

results we find that no two benchmarks reacted the same across all our tests, thus the *optimal design for each benchmark is different*, when factoring in limited resources and power constraints. This also argues for heterogeneous designs which dynamically powers up the appropriate sized cores for efficient execution of each benchmark.

3.4 Related Work

In the following section, we discuss related work that pertains to our limit study. We start by examining past limit studies, and end on a survey of related toolsets that are similar in terms of measuring ILP and TLP.

3.4.1 Past ILP Limit Studies

Historically, many researchers have published ILP limit studies [35, 44, 25, 16, 34, 2, 6], however since the majority of these studies date back several decades, there have been numerous technology advances in the computer architecture field since their publication. Most importantly, none of them have covered the trade-offs and limits between ILP and TLP. In the following subsection, we highlight some significant limit studies.

Wall et al. [44] published one of the first studies of ILP limits with respect to register renaming, branch prediction, loop unrolling, and window size. The chapter concluded that with ideal techniques that are currently available to exploit ILP, parallelism rarely exceeds 5-7. A study by Butler et al. [6] conducted around the same time showed parallelism to be around 17 for the SPEC Suite. Their chapter showed that with optimal hardware design, machines can achieve realistic parallelism of 2-5. A limit study by Austin et al. [2] involved creating single-threaded DFGs similar to our methodology using SPEC benchmarks. They showed ILP ranges from 13 to 23. Austin et al. concluded a very large instruction window is needed to capture the majority of the parallelism. Lam et al. [25] demonstrated that control

flow is a bottleneck in exploiting ILP, our branch prediction experiments reinforce this finding. The authors concluded that, to increase ILP, restrictions on control flow must be eased. Three novel techniques were introduced that helped alleviate control flow, increasing ILP to its full potential: speculative execution, control dependence analysis, and executing various paths in program execution simultaneously. The studies conducted by Wall, Butler, Austin and Lam et al. are the foundation of our limit study, as we seek to replicate their research in a multi-threaded era, where applications now include inter-thread dependencies. The findings in these papers are inconsistent with each other. The ILP limits found in Austin et al.'s were most similar to our finding and coincidentally had a very similar methodology to us. Their and our work both show parallelism to be several orders of magnitude greater than what was found in the other papers. We speculate that the reason why we found the upper bound to be orders of magnitude higher than most work previous is because modern programs are much larger, more complex and operate on much more data allowing for more data-parallel ILP extraction.

Postiff et al. [35] examined ILP in SPEC95 benchmarks. Their approach to find additional ILP was unique compared to previous methods published prior; which involved removing disruptions in the instruction stream caused by the stack pointer. They showed that there is plenty of ILP in an application; however, it is spread out quite expansively. They also concluded that compilers must be involved in mining ILP. This work differs from ours, as Postiff et al.'s work involved modifications to the traces, removing compiler-added code. In contrast we removed only operating system code from our traces. Gonzalez et al. [16] argued that there was a lack of effort to increase ILP by trying to solve the data dependency problem, showing it is a major bottleneck. The solution proposed was to create data speculation techniques to predict values of data dependencies. The study showed that additional ILP could

be exploited by removing pseudo data dependencies and substituting their values with values generated with a prediction algorithm. The paper demonstrated that significant improvements can be achieved by predicting arithmetic data values when using an infinite window. In their work, they noted that speculation techniques would greatly benefit large/infinite instruction windows. Our work agrees with both Gonzales et al. and Postiff et al. [16, 35] findings that ILP is spread out across an application. We show that the majority of ILP can be captured using window sizes greater than 5000 in a few of the benchmarks.

Pai et al. [34] is one of the newer limit studies. The researchers aimed at exploring DLP, since at that time, Single instruction, multiple data (SIMD) had become quite popular. The authors looked into how much DLP is available in a program; as they pointed out, previous studies had not distinguished between TLP and DLP. They found that there is a high degree of available DLP in applications. In our work, we do not directly distinguish between DLP and ILP; we speculate that the majority of ILP captured in our experiments using a window <5000 are a result of DLP.

The most recent study was done by McFarlin et al. [29] in 2013, and is a loosely based limit study that attempts to calculate the upper bounds the OoO Engine performance. Their research focused on the OoO scheduler, and was split into two parts. The first part involves reworking scheduling order of instructions with respect to functional units and operands. Secondly, they looked at scheduling improvements based on hardware speculation support. Based on their study they came up with recommendations on what is needed to have an optimal OoO Engine, such as the need to optimize instructions scheduling statically. They also found that the critical path is highly dependent on load and branch instructions, and addressing these issues can greatly increase performance. In our work, we do not study the impact an OoO

Engine with respect to ILP, as we assume in all our experiments that functional units are an unlimited resource.

We note that many of the past limit studies covered above are somewhat outdated; benchmarks have evolved, and they have been designed to take advantage of TLP. These previous studies workloads used much smaller data sets and different/comparatively less complex algorithms. Further, no previous study examined the relation between TLP and ILP, thread synchronization effects and thread load balancing in multi-threaded applications with respect to the critical path. There is a dearth of studies regarding trade-offs and limits on current benchmarks when it comes to ILP and TLP. These studies need to be updated to reflect behavior of future applications. This is our motivation for pursuing the work in this chapter.

3.4.2 Past TLP Limit Studies

One of the most influential modern studies on the trade-offs of TLP and ILP was written by Hill et al. [19], where Ahmdal's Law was applied to various multicore topologies. Hill et al. developed theoretical mathematical models to determine what type of topology would work best in running multi-threaded applications. Various combinations of dynamic, asymmetric, and symmetric topologies were examined. The work was done at a high level and was designed to stimulate thought rather than provide concrete evidence on the best topologies to utilize when considering multi-threaded applications. In this chapter, we aimed to provide some answers to the questions proposed in Hill et al.'s paper. Esmaeilzadeh et al. [12] modeled multicore scaling limits factoring in single, multi-core, and device scaling for the purpose of measuring speed for parallel applications in the next five generations of technology. Esmaeilzadeh et al. used simplified models and did not fully elaborate on TLP and ILP trade-offs, which is the main focus of this chapter. Our work

differs from theirs as they derived their results and conclusions based on a number of simplified assumptions (the authors modeled characteristics of benchmarks derived from other papers). Our work is distinctive in that we try to reflect real workloads as close as possible using real benchmark traces as the basis of our limit study. In other words instead of modeling their characteristics, we used empirical data to reach our conclusions.

3.4.3 TLP Speculation Techniques

In Section 3.3.2.5 we showed there is significant performance to be gained by removing thread synchronization semantics in the majority of benchmarks analyzed. These results give motivation to revisit old techniques to speculate beyond synchronization primitives as well as motivation to create new techniques. In this section we go over a few techniques in the past that are designed to speculate beyond synchronization primitives gain performance.

Thread Level Speculation (TLS) was introduced in the mid 90s by Steffan et al. [41]. TLS is a technique that generates automatic parallelization of single threaded programs starting at the compiler level. Speculative threads are generated based on the compiler "guessing" whether blocks of codes are independent. These threads are executed at runtime speculatively. Martinez et al. [28], extended TLS to work with parallel applications. The authors argued that many thread synchronization primitives were placed in non-optimal positions in code. Thus, forcing independent code to be dependent due to the constrictions of thread synchronization primitives. Martinez et al.'s work introduced the ability to speculate beyond barriers, locks and flags.

Another TLP speculation technique is Speculative Lock Elision (SPE), developed by Rajwar et al. [36]. SPE detects whether a critical section is truly critical at

runtime. If a critical section is determined to have false inter-thread dependencies the locks are removed and the critical section is executed. Thus, this allows threads not to wait to acquire a lock, therefore reducing execution time of the parallel application.

This Thesis adds to previous work done by showing the upper bound of performance when removing all thread synchronization primitives. The results shown in Section 3.3.2.5 provide motivation to revisit these previous techniques as well as create new ones.

3.4.4 Trace-Driven Tool Sets

To the best of our knowledge, we have built the first simulator that is able to analyze thread and instruction dependencies for benchmarks of current and emerging workloads. There exist a few simulators, however, that share the some of the same characteristics. MaxPar [23] is a simulator developed in 1985 that analyzes data and instruction dependencies in parallel systems. The overall goal of the simulator was to measure the inherent parallelism in parallel applications. Since then, there have been many changes such as underlying ISA, compilers, etc. Our simulator is similar to MaxPar, except to accommodate the benchmarks of today and it can handle inter-thread dependencies. Tasksim [38] is a hybrid simulator that combines both traces and real-time execution of multi-threaded applications. This hybrid system has the ability to generate a single trace and use that trace to run N threads, thus saving time generating traces for given number of cores. The drawback to this simulator is that it does not support many popular parallel languages. They support three languages, with the most notable being OpenMP 3.0. Our simulator is different in that for each set of N threads, you must re-run the traces. Our simulator bypasses high level languages (HLL) and runs at the assembly level; as a result our simulator places no restrictions on HLL or threading model.

3.5 Summary

In this chapter, we conducted a limit study on next-generation multi-threaded benchmarks. We found that there remains a significant amount of ILP in these benchmarks which has yet to be mined. Compared to real machines as much as 929x more ILP is available. We found the upper bound on ILP averaged around 200 instructions/cycle for all benchmarks, far exceeding current high-performance processor cores. Much of this ILP, however, is much further than 5000 instructions away. The plethora of ILP found should be motivation for the computer architecture community to revisit old techniques as well attempt to create new techniques to extract this ILP. As TLP increases, there is often a trade off in ILP, and it can decrease as much as 45%, depending on the complexity of the parallel algorithm. We also found that thread dependencies had a detrimental effect on cycle time, increasing it as much 9x. Adding realistic branch prediction, and realistic memory latency resulted in ILP degradation of 67% and 31% on average respectively. From this study it is clear that there are large performance improvements to be had when it comes to next-generation parallel workloads. In particular, the results of this study argue that research into more aggressive thread synchronization speculation techniques can have a significant performance impact. They also argue that research into run-time critical path identification will be critical to the success of asymmetric multi-core designs. From the all the experiments that we conducted, there were no two benchmarks that reacted the same way to all our tests, leading us to believe that the optimal design for each benchmark is different, with respect to performance and power.

4. SHARED-MEMORY CHARACTERIZATION ANALYSIS

One important unanswered question we can can solve using our novel simulator DotSim is: what is the potential benefit of applying latency reducing techniques to shared memory in critical sections with respect to execution time and ILP? To the best of our knowledge there has yet been a study done answering this question using shared memory multi-threaded applications. We conduct this study using our novel simulator DotSim, and the PARSEC benchmark suite.

In this chapter, we conduct an idealized workload characterization study, focusing on the truly shared loads between threads in multi-threaded benchmarks. We define truly shared loads to be when a load is directly dependent on a store from another thread. This workload characterization study is designed to answer what performance benefit is there into using latency reducing techniques such as prefetching shared memory within a critical section. We do this by calculating the degree of criticality of truly shared loads between threads by speeding up accesses to them and observing the effects on execution time, critical path and ILP.

This chapter is a direct contrasts from the work done in Section 3.3.2.5 were we ignored thread synchronization primitives (but respected memory consistency). We showed in Section 3.3.2.5, that removing thread synchronization primitives resulted in a significant increase on the upper bound of performance. In this workload characterization study we perform several experiments using DotSim:

1. *We quantify the amount of sharing done between threads, with respect to the overall program and the critical path.*

2. *Provide a visual view of sharing between threads, in attempt to see if their is exists an exploitable pattern to gain performance and reduce power.*

3. *We drastically speed up and slow down the latency of truly shared memory between threads and observe changes in ILP and execution time.*

4. *We add Miss Status Handle Registers (MSHRs) to our cache model that was introduced in Section 3.3.2.6. We observe the impact on ILP, CP and execution time, repeating the experiments above.*

In our study we find that on average, true sharing among benchmarks is insignificant compared to all memory access except for one benchmark. We also find that truly shared memory does not affect the critical path under our given constraints. Therefore on average there is minimal impact on execution time, and the maximum impact on the upper bound of performance improvements on execution time and ILP is $< 1\%$. However, it is worth noting that these results are highly dependent on the benchmarks, of which we used PARSEC benchmark suite. These results are unclear across all shared-memory multi-threaded applications.

4.1 Motivation

There has been much work performed in attempting to exploit critical sections of parallel code [7, 43, 8, 24, 37, 28, 11, 42, 36]. These techniques include allocating additional computing resources, speculating beyond critical sections, lock prediction, and reducing coherency related traffic. Our work in Section 3.3.2.5, showed there is up to 80% performance improvement possible speculating beyond critical section. However, this experiment did not quantify the impact that truly shared memory has on ILP and execution time. We think it is worth pursuing quantifying the degree of criticality of truly shared memory. This will provide an upper bound on performance that possibly could motivate potential latency reducing techniques such as prefetching shared data to improve multi-threaded application performance. These potentially latency reducing techniques could lead to reduction in the length

of the critical path, and possibly increase in ILP. To the best out our knowledge no such study has been recently done. Trancoso et al. [43] produced a very similar study using a distributed memory system configuration with Splash 2 benchmarks using an in-order machine that is very sensitive to memory latency. Additionally, their study did not focus on conducting a limit study on the upper bound of performance like we determine in this chapter.

4.2 Methodology

In this section we discuss the methodology of conducting a shared-memory workload characterization study on multi-threaded applications. First, we generated dynamic instruction stream traces using the gem5 Simulator [5] and the PARSEC Benchmark Suite [4] compiled for the Alpha ISA [15]. All the PARSEC benchmarks that our simulation infrastructure supports are presented.

Once traces were generated, we pre-process them by identifying memory dependencies inter and intra thread, as discussed in previous chapters. An individual trace set was generated for N_{min}=64. These traces were then fed into our off-line analysis tools which we used to conduct our workload characterization study. We continue to use the small input size to generate traces. We also stick to focusing on only instructions in the Region of Interest (ROI). In order to ensure that the integrity of this workload characterization study was not effected by input size we selected a few benchmarks to run using small, medium and large input sizes. The results were nearly identical for all input sizes. In the subsection below we discuss how the workload characterization study was conducted.

4.2.1 Workload Characterization Study

The objective of our experiments is to measure the amount of true sharing done and to measure the degree of criticality of truly shared memory. We define true

sharing to be when a load is directly dependent on a store from another thread. In order to do this we counted the number of loads from one thread that depended on a store from another thread. We kept track of this information with respect to the CP (as discussed in Chapter 3) as well as the overall program. In addition we kept track of which threads shared with other threads using a matrix. Finally, we also kept track of overall memory accesses (L1, L2, L3, and DRAM) with respect to the benchmark as a whole and its CP.

In order to determine the degree of criticality of truly shared memory we sped up and slowed down memory accesses 150X and 10X respectively then compared it to our baseline configuration. A slow down of 10X represents a worst case scenario for a cache coherency penalty. We used DotSim to simulate these experiments using our cache simulator discussed in Section 3.3.2.6. For our realistic memory system we implemented simple cache hierarchy, modeling private L1 and L2 caches and a shared L3 last-level cache. The sizes of each cache are 256KB, 512KB, and 64MBs respectively. The three levels of the cache have an access delay of 1, 10, 30 respectively, a L3 cache miss results in a 150 cycle delay. We modeled any load that required access to a store in another thread of having an access delay of 150 to represent coherency traffic. We measured the degree of criticality by observing the effects on execution time as we changed access time to truly shared memory from 1 cycle (speed up of 150X), and 1500 cycles (slow down of 10X) with a baseline latency of 150 cycles. In our DotSim setup, we enabled thread synchronization primitives, enabled branch prediction, and a 128 instruction window with an issue width of 4. We then repeat the experiments adding the effects of Miss Handle Status Registers (MSHRs) to our cache model with a maximum 8 outstanding misses for each level of cache.

4.3 Evaluation

In our workload characterization study we answer the following questions in this section:

1. *How much of memory traffic is made up of truly shared loads between threads?*
2. *What is the degree of criticality of truly shared loads?*
3. *What is the impact on ILP and execution time when adding Miss Status Handle Registers (MSHRs) to our cache model?*
4. *What are possible sharing patterns that could potentially be exploited with respect to power and performance?*

We believe that answers these questions will either argue against or for latency reducing techniques for truly shared loads as well as possible hints into exploiting memory traffic in terms of optimizing power and performance.

4.3.1 How Much of Memory Traffic is Made Up of Truly Shared Loads Between Threads?

In this experiment, we keep track of all loads within the overall program and CP. We keep track of all loads by gathering statistics on what level of cache hierarchy does each loads hit. What is unique about this experiment is we count the number of loads that are truly shared. We define truly shared loads to be when a load is directly dependent on a store from another thread. Truly shared loads are what thread synchronization primitives are designed to to preserve in terms of memory ordering and consistency. In the case of truly-shared memory we count it as *shared load*, and do not count this towards hitting any other level of the memory hierarchy.

Figure 4.1 shows the breakdown of memory traffic. For Blackscholes we see there is no sharing, we find this to be consistent with previous work which has shown there is no sharing between threads [4]. On average, for all benchmarks there is 5% memory traffic related to truly shared loads among threads. Accesses to L3 and DRAM accesses are insignificant for all benchmarks. Its worth nothing that this is a result of truly shared loads not counting towards hitting L3 and DRAM (such as L3 and DRAM). Bodytrack has the most truly shared loads. Overall, truly shared loads make up nearly 20% of memory accesses of Bodytrack, but its critical path only makes up 7% of truly shared loads. This implies that sharing among threads is not likely a bottleneck on the critical path. We are unsurprised by the amount of shared between threads in Bodytrack as we noted in Section 3.2.2.4, that there was a high degree of thread synchronization primitives, which is likely correlated to prevent threads from overriding each other's critical section and preserving data memory consistency.

4.3.2 What is the Degree of Criticality of Truly Shared Loads?

On average the number of truly shared loads represents a small portion of the overall program and CP. However, what is pivotal is how critical truly shared memory is even if it makes up a small portion of memory traffic. If indeed it is critical, then that would provide enough motivation to attempt to reduce latency of these accesses in hopes of increasing execution time and ILP. In this experiment, we slow down truly shared loads by $10X$. We then observe the impact on execution time and CP ILP. The degree of criticality is determined by the impact of speeding up and slow down truly shared memory has on CP ILP and execution time. We first measure the degree of criticality using our setups discussed in Section 4.2 without using MSHRs. We find that the effects on CP ILP and execution time for speeding up as well as slow down

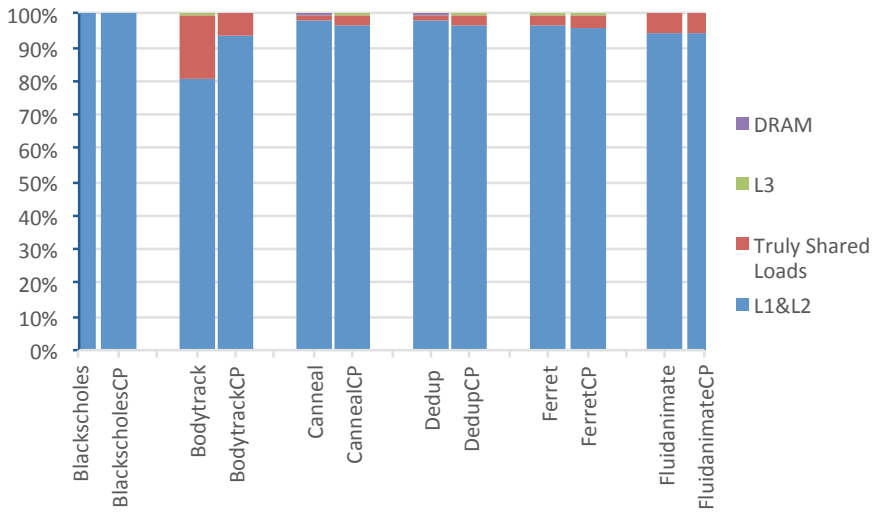


Figure 4.1: Breakdown of memory accesses of the CP and overall program.

was insignificant (<1%) for **all benchmarks**. We find this somewhat unsurprising as on average, truly shared memory makes up 5% of all memory accesses.

We hypothesize that the lack of criticality of truly shared memory could be possibly due to the lack of additional details in our modeling of the cache memory system. We then add MSHRs, to reduce the amount of outstanding misses that could be possible in each level of cache to eight. Thus, this addition reduces the amount of ILP possible, as we attempt to make our configuration more dependent on the memory system. We then extend our experiments from Chapter 3 in Section 3.3.2.6, where we observe the impact on performance adding a realistic memory system. In Figure 4.2 we see the impact of adding MSHRs to the memory we system. We note that Canneal (which was previously identified as the most sensitive to adding a memory system) has the worst impact when it comes to ILP. Bodytrack has the worst impact when it comes to execution time. On average adding MSHRs impacts ILP and execution times by 12%, were all benchmarks were affected except for Blackscholes

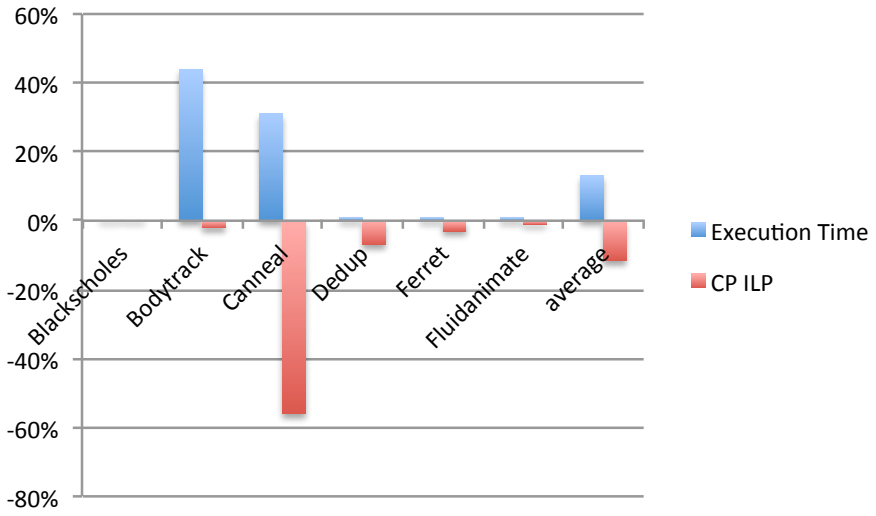


Figure 4.2: Changes in ILP_{MT_CP} and execution time when adding MSHRs normalized without MSHRs.

which is heavily dependent on floating point calculations and not memory.

Using our cache model incorporating MSHRs, we then repeat slowing down and speeding up truly shared loads. We then measure the degree of criticality. We show the results in Figure 4.3. The degree of criticality of these truly shared loads, are still insignificant, having an upper bound performance benefits of 1% (much higher than without MSHRs were benefits were <1%). We can conclude from these results based on our constraints and benchmarks that we tested, truly shared loads are not critical and have very minor effects on ILP and execution time. These results support the performance gains we saw in Section 3.3.2.6 were we removed thread synchronization primitives and saw significant gains in performance. We believe the results here show that truly shared loads are not the performance critical part of of an application.

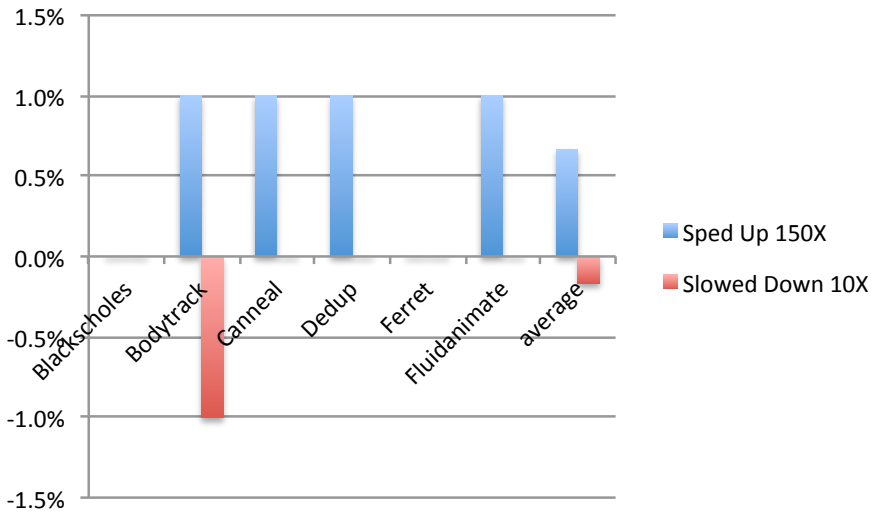


Figure 4.3: Changes in ILP_{MT_CP} and execution time when speeding up and slowing down by a 150X using MSHRs in our cache model normalized against our default latency model.

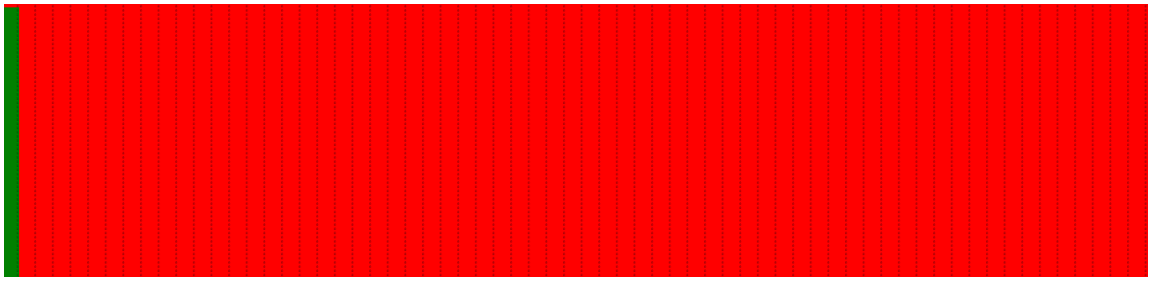
4.3.3 What are possible sharing patterns that could potentially be exploited in terms of power and performance?

In our final experiment, we provide a graphical visual representations of truly shared memory between threads using a heat map. We use a matrix to count the number of stores supplied to one thread that is consumed by a load in another thread. Columns represents the thread providing the store, were the row represents the thread requiring a store for its dependent load. Columns and rows are ordered in terms of Thread ID, where Thread IDs are assigned chronologically and are allocated during thread spawning phase. Thread 0 is always the main thread, which often spawns all the other threads (not in all cases). Figure 4.4 and Figure 4.5 show all six heat maps for all benchmarks. We note that despite setting N_{min}=64 as stated in Section 3.3.2.1, benchmarks often spawn more than 64. Therefore the heat maps are not all the same size. These heat maps represent the frequency of thread

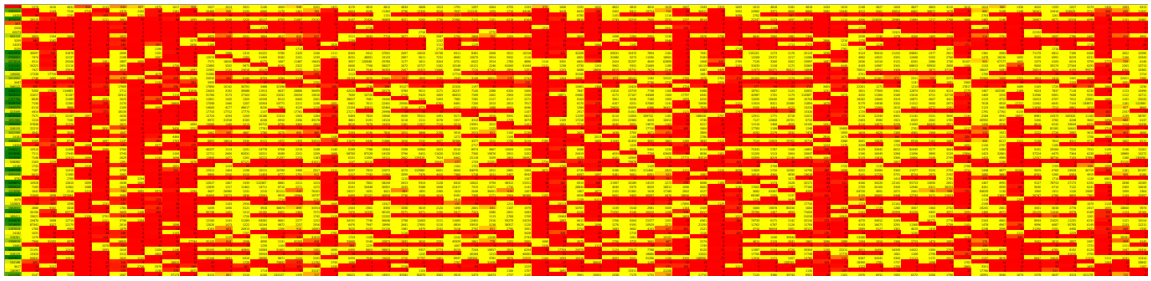
communications via the store-load producer consumer relationship. Red represents the lowest value, while yellow represents the midpoint threshold and green represents the highest values thorough the colored matrix.

For Blackscholes we see no sharing as expected. For Bodytrack we see that the main thread (Thread 0) does the most sharing. Thread 0 seems to be the producers while a lot of the other threads consume off of it. It would be ideal to place Thread 0 to be the middle core in the CMP design, therefore reducing the distance when communicating to other cores, This would reduce latency communications and power consumption. Canneal, has an interesting sharing diagonal pattern where each thread, shares with the thread directly next to it, in a tightly producer-consumer relationship. In the case of Canneal, threads should be placed next each other via spawn order, since each thread often blocks the thread before it. For Dedup there is a very obvious pattern, threads tend to share with neighbor threads in square clusters (note the yellow squares on each corner of the map). This means that thread should be placed in order by when they spawn, much like our suggestions for Canneal. Fluidanimate has a very similar pattern to that of Canneal. Finally, Ferret also has a pronounced pattern of small sections of thread sharing between ordered threads, as well as Thread 2 producing data that is required by a lot of other threads that are spread out. This implies that Thread 2 should be placed in such a way that all other threads can reach it the quickest.

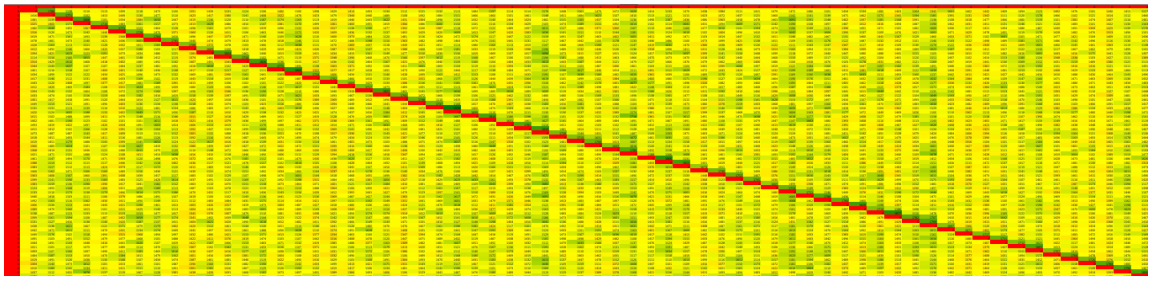
From these heat maps we can conclude that random thread scheduling and placement is non optimal. Thread scheduling is important, which should be used as an advantage to shorten communication distance and reduce latency. Optimizing thread scheduling moving communicating pairs of threads which includes thread placement with CPU cores and mapping cache lines in the last level cache (possibly optimizing via page coloring) to match the thread core location.



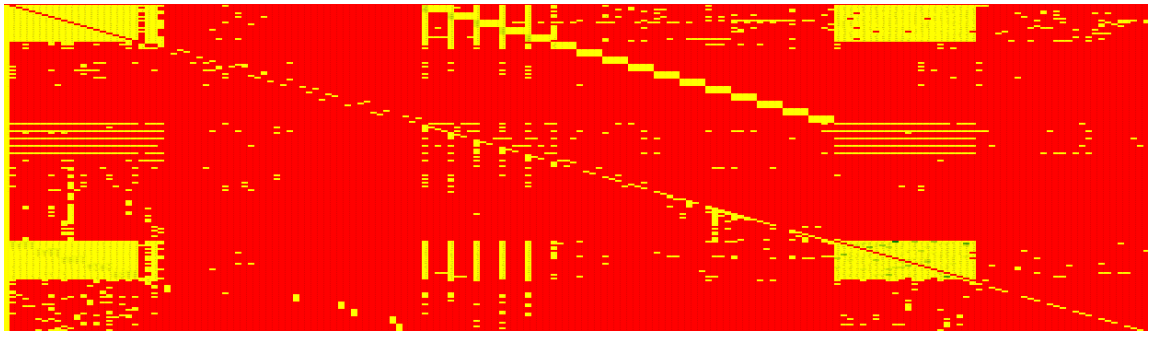
(a) Blackscholes



(b) Bodytrack

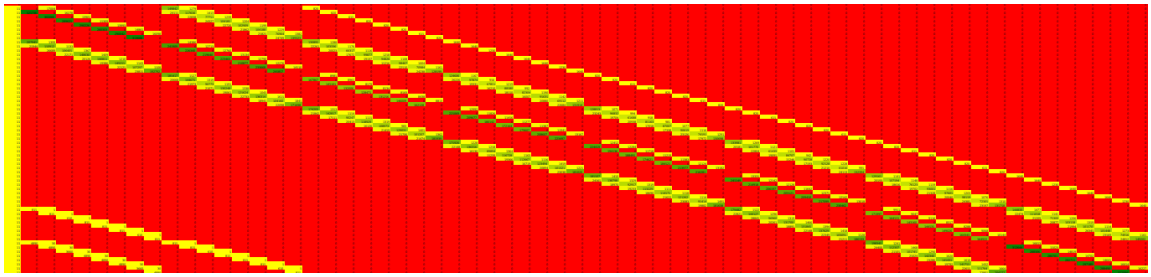


(c) Canneal



(d) Dedup

Figure 4.4: Heat maps representing thread to thread communications.



(a) Fluidanimate

(b) Ferret

Figure 4.5: Heat maps continued.

4.3.4 Inaccuracies In Our Studies

Other studies discussed in the related work section have found performance improvements in coherence protocol optimizations. We speculate that the differences in our study and previous studies is that we do not model interconnection networks, bus and coherency traffic which when combined could possible increase the degree of criticality of truly shared memory. Therefore, future work would involve adding a more realistic interconnection network, and produce synthetic coherency traffic from our traces. However, we emphasize the goal of this study was to provide a upper bound on shared memory.

4.4 Related Work

Transcoso and Torrellas performed a very similar study using a subset of Splash benchmarks, where they sped up critical sections by prefetching shared data, and other variants and tweaks designed to reduce latency time [43]. They found impressive performance improvements with their techniques sometimes achieving greater than 50% reduction in execution time. However, their methodology was much different than ours, specifically since they used an in-order machine which is very sensitive to memory latency changes in terms of instructions per cycle (IPC). This indicates that perhaps out-of-order machines greatly reduce the impact of shared memory prefetching.

Demetriades [8] et al. published a paper on predicting coherence communications, thus resulting in reduced latency misses. They created a way to predict coherency communication thereby, improving average latency times in directory protocols. They tested Splash-2 and PARSEC benchmarks. A few of their benchmarks were identical to ours, however their results do not match ours. The reasoning is we used a simplified modeled, were do not model coherency traffic, protocols or any interconnection networking details. These additional details are likely making our results inconsistent. Another reason for the discrepancy is we focused on true sharing of data between threads, but they focused on the overall critical section. There have been many other coherency prediction mechanisms that attempt to reduce the amount of latency required to communicate/exchange shared data [24, 37]. Many of the studies were done either using Distributed Shared Memory (DSM) which has orders of magnitude greater latency than our configuration of studying CMPs. Many of these studies also used an interconnection network with latency having to factor in bus contention. Therefore, comparing to our somewhat oracle study to be much

different as well as a using different benchmarks.

Cheng et al. proposed varying interconnect wires that have different latency, bandwidth and energy properties. They proposed that coherency communications on the interconnection networks be transferred via reduced latency wires that have much lower bandwidth [7]. There results show performance improvements of 11.2%. There work is an alternative way to try to reduce the amount of sharing communication done via computing nodes. However, the work done in this paper varies vastly different than what we have done here. As the amount of realism and detail implemented in their simulations is much greater than ours, as we previously discussed in this section.

All these papers covered here attempt to reduce the amount of coherency latency done with the requirements of sharing data among cores. They often show very realistic ways of trying to improve performance. The results shown in these paper is different from our methodology as we have implemented a much simpler model, measuring the performance benefits of truly shared data. Ultimately, what we can conclude from our study is that the critical path of the benchmarks under test are definitely not bottlenecked by truly shared loads among threads.

4.5 Summary

In this chapter we present a workload characterization study on sharing between threads in multi-threaded applications. Overall, we show that the amount of true sharing among threads is quite low, aside from one benchmark (Bodytrack). These results supports our previous results of removing thread synchronization primitives resulted in significant performance improvement. Most importantly we show that the upper bound of performance on degree of criticality of true sharing is $< 1\%$, and even then its only impactful on two benchmarks. Thus we conclude based on these

results focusing on latency reducing techniques to prefetch truly shared data is not justified but focusing on the overall critical section is.

5. CONCLUSIONS

In this dissertation, we made three contributions. (1) We develop DotSim, a trace-driven tool kit that is designed to explore the limits of instruction- and thread-level scaling and identify microarchitectural bottlenecks in multi-threaded applications. DotSim excels at first-order modeling of many novel microarchitectural approaches. DotSim is also ideal for validating other simulators with less abstraction. (2) We used DotSim to conduct a limit study on next-generation multi-threaded benchmarks. We found that there remains a significant amount of ILP in these benchmarks which has yet to be mined. Compared to real machines as much as 929x more ILP is available. We found the upper bound on ILP averaged around 200 instructions/cycle for all benchmarks, far exceeding current high-performance processor cores. Much of this ILP, however, is much further than 5000 instructions away. The plethora of ILP found should be motivation for the computer architecture community to revisit old techniques as well attempt to create new techniques to extract this ILP. It should be noted the calculating the upper bound on ILP is dependent on compiler optimization and underlying ISA. Compiling with optimization is likely to reduce the amount of ILP versus un-optmized code as optimized code is more efficient in reducing the amount of instruction (such as dead block elimination). The underlying ISA may also have an affect on ILP due to the varying amount of registers ISA's have. As a reduced amount of registers likely forces register values to spill into memory thus serializing instructions (adding inter-thread store to load dependencies) thus reducing the amount of ILP compared to an ISA with more registers. As TLP increases, there is often a trade off in ILP, and it can decrease as much as 45%, depending on the complexity of the parallel algorithm. We also found that thread dependencies had a

detrimental effect on cycle time, increasing it as much 9x. Adding realistic branch prediction, and realistic memory latency resulted in ILP degradation of 67% and 31% on average respectively. From the all the experiments that we conducted, there were no two benchmarks that reacted the same way to all our tests, leading us to believe that the optimal design for each benchmark is different, with respect to performance and power as we show with Table 5.1.

Finally (3) we present a workload characterization study on sharing between threads in multi-threaded applications. Overall, we show that the amount of true sharing among threads is quite low. We conclude that based on the results of our study, focusing on latency reducing techniques to prefetch truly shared data is not justified. Lastly in the study we show that random pin of cores to threads is not an ideal solutions, as there are exploitable memory sharing patterns.

5.1 Future Work

This dissertation covered a high level view of studying the limits of next generation multi-threaded benchmarks with respect to micro-architecture design. The goal of future work should be to add layers of details in micro-architectural design, removing the layers of abstractions we have provided in DotSim. Example of additional micro-architecture details are to model an interconnection network, realistic branch predictors, and Out-of-Order execution engine. Once additional micro-architectural details have been added the experiments conducted here at the very least should be repeated. DotSim's front end should also be changed to accept X86 ISA. Lastly experiments should be conducted that involve speeding up threads that make up critical path, which we identified in Chapter 3's final experiment, observing impact on execution time and ILP.

Benchmark	Max ILP	Increasing TLP	Threading Inefficiency	Enabling Thread Semantics	Optimal Instruction Window	Cache or Branch Predictor
Blackscholes	326	Flat	Linear	No Effect	5K	BP
Bodytrack	929	Increases	Super Linear	Huge Effect	NA	BP
Canneal	75	Decreases	Linear	Huge Effect	1024	Cache
Dedup	29	Decreases	Super Linear	No Effect	128	BP
Ferret	31	Decreases	Super Linear	Huge Effect	512	BP
Fluidanimate	77	Increases	Linear	Effect on N>32	5K	BP
Vips	88	NA	NA	NA	512	NA
X264	100	NA	NA	NA	512	NA

Table 5.1: Summary of results for each benchmark.

REFERENCES

[1] Sebastian Anthony. Intel: Haswell will draw 50% less power than Ivy Bridge. http://www.extremetech.com/computing/156739-intel-haswell-will-draw-50-less-power-than-ivy-bridge, 2013.

[2] Todd M. Austin and Gurindar S. Sohi. Dynamic dependency analysis of ordinary programs. In *the 19th Annual International Symposium on Computer Architecture (ISCA)*, pages 342–351, 1992.

[3] Major Bhadauria, Vincent M. Weaver, and Sally A. McKee. Understanding parsec performance on contemporary cmps. In *the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 98–107, 2009.

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *The 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *ACM Computer Architecture News*, 39(2):1–7, May 2011.

[6] Michael Butler, Tse-Yu Yeh, Yale Patt, Mitch Alsup, Hunter Scales, and Michael Shebanow. Single instruction stream parallelism is greater than two. In *the 18th Annual International Symposium on Computer Architecture (ISCA)*, pages 276–286, 1991.

[7] Liqun Cheng, Naveen Muralimanohar, Karthik Ramani, Rajeev Balasubramonian, and John B. Carter. Interconnect-aware coherence protocols for chip multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 339–351, Washington, DC, 2006. IEEE Computer Society.

[8] S. Demetriades and Sangyeun Cho. Predicting coherence communication by tracking synchronization points at run time. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pages 351–362, Dec 2012.

[9] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[10] DOT. Dot (graph description language).

[11] Kristof Du Bois, Stijn Eyerman, Jennifer B. Sartor, and Lieven Eeckhout. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. In *the 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 511–522, 2013.

[12] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *the 38th annual international symposium on Computer architecture (ISCA)*, pages 365–376, 2011.

[13] K. Ghose et al. Marssx86: Micro architectural systems simulator, 2012.

[14] Ehsan Fatehi and Paul Gratz. Ilp and tlp in shared memory applications: A limit study. In *Proceedings of the 23rd International Conference on Parallel*

Architectures and Compilation, PACT '14, pages 113–126, New York, NY, 2014. ACM.

[15] Mark Gebhart, Joel Hestness, Ehsan Fatehi, Paul Gratz, and Stephen W. Keckler. Running PARSEC 2.1 on M5. Technical report, The Univ. of Texas at Austin, Dept. of Comp. Sci., 2009.

[16] José González and Antonio González. Limits of instruction level parallelism with data speculation. In *Proc. of the VECPAR Conf*, pages 585–598. Citeseer, 1998.

[17] Z. Guz, O. Itzhak, I. Keidar, A. Kolodny, A. Mendelson, and U.C. Weiser. Threads vs. caches: Modeling the behavior of parallel workloads. In *Computer Design (ICCD), 2010 IEEE International Conference on*, pages 274–281, Oct 2010.

[18] Zvika Guz, Evgeny Bolotin, Idit Keidar, Avinoam Kolodny, Avi Mendelson, and Uri C. Weiser. Many-core vs. many-thread machines: Stay away from the valley. *IEEE Comput. Archit. Lett.*, 8(1):25–28, January 2009.

[19] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

[20] Aamer Jaleel, Robert S. Cohn, Chi keung Luk, and Bruce Jacob. Cmp$im: A binary instrumentation approach to modeling memory behavior of workloads on cmps. Technical report, University of Maryland, 2006.

[21] D.A. Jimenez. Piecewise linear branch prediction. In *the 32nd International Symposium on Computer Architecture (ISCA)*, pages 382–393, June 2005.

[22] Daniel A. Jimenez. An optimized scaled neural branch predictor. In *the 2011 IEEE 29th International Conference on Computer Design (ICCD)*, pages 113–

118, 2011.

[23] Ding kai Chen. Maxpar: An execution driven simulator for studying parallel systems. Technical report, University of Illinois at Urbana-Champaign, 1989.

[24] An-Chow Lai and Babak Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 139–148, New York, NY, 2000. ACM.

[25] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. In *the 19th Annual International Symposium on Computer Architecture (ISCA)*, pages 46–57, 1992.

[26] Gabriel H. Loh, Samantika Subramaniam, and Yuejian Xie. Zesto: A cycle-level simulator for highly detailed microarchitecture exploration. In *ISPASS*, pages 53–64. IEEE, 2009.

[27] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, 2005. ACM.

[28] José F. Martínez and Josep Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. *SIGOPS Oper. Syst. Rev.*, 36(5):18–29, October 2002.

[29] Daniel S. McFarlin, Charles Tucker, and Craig Zilles. Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism? In *the*

Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 241–252, 2013.

[30] Matteo Monchiero, Jung Ho Ahn, Ayose Falcón, Daniel Ortega, and Paolo Faraboschi. How to simulate 1000 cores. *SIGARCH Comput. Archit. News*, 37(2):10–19, July 2009.

[31] G.E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, April 1965.

[32] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[33] Tony Nowatzki, Jai Menon, Chen-Han Ho, and Karu Sankaralingam. gem5, GPGPUsim, McPAT, GPUWattch, <Your favorite simulator here> Considered Harmful. In *11th Annual Workshop on Duplicating, Deconstructing and Debunking*, 2013.

[34] Sreepathi Pai, R Govindarajan, and MJ Thazhuthaveetil. Limits of data-level parallelism. *14th Annual IEEE International Conference on High Performance Computing*, December 2007.

[35] Matthew A. Postiff, David A. Greene, Gary S. Tyson, and Trevor N. Mudge. The limits of instruction level parallelism in spec95 applications. *SIGARCH Comput. Archit. News*, 27(1):31–34, March 1999.

[36] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *the 34th Annual ACM/IEEE International Symposium on Microarchitecture (Micro)*, pages 294–305, 2001.

[37] Ravi Rajwar, Alain KŁgi, and James R. Goodman. Inferential queueing and speculative push, 2003.

[38] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, and M. Valero. Trace-driven simulation of multithreaded applications. In *the 2011 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 87 –96, april 2011.

[39] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. *SIGARCH Comput. Archit. News*, 41(3):475–486, June 2013.

[40] André Seznec. A new case for the tage branch predictor. In *the 44th Annual IEEE/ACM International Symposium on Microarchitecture (Micro)*, pages 117–127, 2011.

[41] J. Steffan and T Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *the 4th International Symposium on High-Performance Computer Architecture (HPCA)*, 1998.

[42] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. *SIGPLAN Not.*, 44(3):253–264, March 2009.

[43] P. Trancoso and J. Torrellas. The impact of speeding up critical sections with data prefetching and forwarding. In *Parallel Processing, 1996. Vol.3. Software., Proceedings of the 1996 International Conference on*, volume 3, pages 79–86 vol.3, Aug 1996.

[44] David W. Wall. Limits of instruction-level parallelism. In *the fourth international conference on Architectural support for programming languages and*

operating systems (ASPLOS), pages 176–188, 1991.

[45] M.T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator". *ISPASS*, 2007.