ABSTRACT

Title of dissertation: EFFICIENT MULTIPROGRAMMING FOR MULTICORES WITH SCAF

Timothy Mattausch Creech
Doctor of Philosophy, 2015

Dissertation directed by: Professor Rajeev Barua
Department of Electrical and Computer Engineering

As hardware becomes increasingly parallel and the availability of scalable parallel software improves, the problem of managing multiple multithreaded applications (processes) becomes important. *Malleable* processes, which can vary the number of threads used as they run [1], enable sophisticated and flexible resource management. Although many existing applications parallelized for SMPs with parallel runtimes are in fact already malleable, deployed run-time environments provide no interface nor any strategy for intelligently allocating hardware threads or even preventing oversubscription. Prior research methods either depend upon profiling applications ahead of time in order to make good decisions about allocations, or do not account for process efficiency at all, leading to poor performance. None of these prior methods have been adapted widely in practice. This paper presents the **Sc**heduling and **A**llocation with **F**eedback (SCAF) system: a drop-in runtime solution which sup-

ports existing malleable applications in making intelligent allocation decisions based on observed efficiency without any changes to semantics, program modification, offline profiling, or even recompilation. Our existing implementation can control most unmodified OpenMP applications. Other malleable threading libraries can also easily be supported with small modifications, without requiring application modification or recompilation.

In this work, we present the SCAF daemon and a SCAF-aware port of the GNU OpenMP runtime. We present a new technique for estimating process efficiency purely at runtime using available hardware counters, and demonstrate its effectiveness in aiding allocation decisions.

We evaluated SCAF using NAS NPB parallel benchmarks on five commodity parallel platforms, enumerating architectural features and their effects on our scheme. We measured the benefit of SCAF in terms of sum of speedups improvement (a common metric for multiprogrammed environments) when running all benchmark pairs concurrently compared to equipartitioning — the best existing competing scheme in the literature. If the sum of speedups with SCAF is within 5% of equipartitioning (i.e., improvement factor is 0.95X $<$ improvement factor in sum of speedups $<$ 1.05X), then we deem SCAF to break even. Less than 0.95X is considered a slowdown; greater than 1.05X is an improvement. We found that SCAF improves on equipartitioning on 4 out of 5 machines, breaking even or improving in 80-89% of pairs and showing a mean improvement of 1.11-1.22X for benchmark pairs for which it shows an improvement, depending on the machine.

Since we are not aware of any widely available tool for equipartitioning, we

also compare SCAF against multiprogramming using unmodified OpenMP, which is the only environment available to end-users today. SCAF improves or breaks even on the unmodified OpenMP runtimes for all 5 machines in 72-100% of pairs, with a mean improvement of 1.27-1.7X, depending on the machine.

EFFICIENT MULTIPROGRAMMING FOR MULTICORES WITH SCAF

by

Timothy Mattausch Creech

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2015

Advisory Committee:
Professor Rajeev Barua, Chair/Advisor
Professor Donald Yeung, Co-Advisor
Professor Bruce Jacob
Professor Manoj Franklin
Professor Peter Keleher, Dean's Representative

Acknowledgments

I would like to express my sincere gratitude to my advisor, Prof. Rajeev Barua, for his guidance, patience, and enthusiasm throughout my graduate studies. Whether I was stuck on a problem or losing focus, our discussions were always the key to progress. His advice throughout the years has certainly shaped and improved me as a writer and researcher. It is no overstatement to say that this work would not have been possible without his support.

I would also like to thank Prof. Donald Yeung, Prof. Bruce Jacob, Prof. Manoj Franklin, and Prof. Peter Keleher for their guidance and feedback. Large portions of this work are directly resultant from this advice, and represent significant improvements.

This work was supported by a NASA Office of the Chief Technologist's Space Technology Research Fellowship (Grant NNX11AM99H). As a result, I was able to work with many fine people at the NASA Goddard Space Flight Center and the Ames Research Center. Specifically, I would like to thank Dan Mandl, Vuong Ly, and Don Sullivan, among others, for their support and advice throughout the years. The compute resources made available to me through the NSTRF program were simply invaluable, as was the experience of working alongside these engineers.

I would also like to thank the University of Maryland's Institute for Advanced Computer Studies (UMIACS) for providing access to additional compute resources. I thank my labmates and friends in A.V. Williams—Jounghoon, Mark, Aparna, Kapil, Khaled, Fady, Kyungjin, Alex, Danny, Julien, and Matt—for their companionship and advice over the years.

I thank Karen Shih, whose friendship and encouragement have been a tremendous influence throughout both my undergraduate and graduate studies.

Finally, I can't thank my parents, Jay and Laura Creech, enough. They have always encouraged my education, and have been a continuous source of support and inspiration.

Contents

List of Tables

List of Figures

List of Abbreviations

API	Application Program Interface
CG	Conjugate Gradiant (an NPB benchmark)
CMT	Clustered Multithreading
COW	Copy on Write
DDC	Dynamic Distributed Cache
EQ	Equipartitioning
FT	Fast Fourier Transform (an NPB benchmark)
GNU	GNU's Not Unix (recursive)
IPC	Instructions per Cycle
LU	Lower-Upper symmetric Gauss-Seidel (an NPB benchmark)
NAS	NASA Advanced Supercomputing
NASA	National Aeronautics and Space Administration
NPB	NAS Parallel Benchmarks
NUMA	Non-Uniform Memory Architecture
OS	Operating System
PAPI	Performance Application Programming Interface
POSIX	Portable Operating System Interface
RC	Resistor-Capacitor (circuit)
SCAF	Scheduling and Allocation with Feedback
SMP	Symmetric Multiprocessing
SMT	Simultaneous Multithreading
SP	Scalar Pentadiagonal (an NPB benchmark)
SUIF	Stanford University Intermediate Format
TBB	Thread Building Blocks
TLB	Translation Look-aside Buffer

UA	Unstructured Adaptive (an NPB benchmark)
UMIACS	University of Maryland Institute for Advanced Computer Studies
UNIX	not an acronym; a family of operating systems
VM	Virtual Memory; sometimes Virtual Machine
VMM	Virtual Memory Manager

Chapter 1

Introduction

1.1 Problem Description and Overview

When running multiple parallelized programs on many-core systems, such as Tilera's TilePro64, or even large Intel x86-based SMP systems, a problem becomes apparent: each application makes the assumption that it is the only application running; therefore parallel applications generate enough threads to use the entire machine, and consequently the machine is quickly oversubscribed if multiple programs are running. A machine is said to be oversubscribed when the number of computationally intensive threads exceeds the number of available hardware contexts. When the number of cores is at least equal to the number of time-intensive applications, then space-sharing the cores between those applications becomes an option. With space-sharing, each process uses some portion of the cores exclusively, avoiding oversubscription. However, in practice this is rarely done on interactive systems. Modern operating systems attempt to time-share the hardware resources by context switching, but this is a poor solution. The context switching incurs additional overhead as the kernel must save and restore the state of the hardware context. Caches may be repeatedly refilled as access patterns oscillate. Further, when some threads participating in a barrier are context-switched out, other threads in the same barrier may incur long waiting times, reducing system throughput. Finally, some synchronization techniques, such as spinlocks, depend heavily on the presence of ded-

icated hardware contexts for reasonable performance. When each process is capable of fully loading the machine, space sharing is an efficient and simple alternative to fine-grained time-sharing. A more detailed discussion of other alternatives as well as justification for choosing space sharing can be found in chapter 2.

In this paper, we define the parallel efficiency of executed code to be $E \equiv \frac{S}{p}$, where executing the code in parallel on p hardware contexts yielded a speedup S over serial execution. We say that a multiprogrammed parallel system has high total efficiency when the sum of speedups achieved by its processes is high. That is, the average hardware context contributes a large speedup.

When dealing with truly malleable parallel programs, the ideal solution is to actually change the number of software threads that need to be scheduled, approximating space sharing. We argue that this should be done automatically and transparently to the system's users. Existing parallelization runtimes generally allow users to specify the maximum number of threads to be used at compile-time or run-time. However, this number remains fixed for the duration of the program's execution, and it is unreasonable to expect users on a system to manually coordinate. Furthermore, even on a single-user system it is not always clear how best to allocate hardware contexts between applications if good total system efficiency is desired: scalability may vary per application and across inputs. Finally, in order for a solution to be adopted, it should not require new programming paradigms or place excessive demands on the users. That is, users should not be required to rewrite, recompile, profile, or spend time carefully characterizing their programs in order to benefit. We believe that this cost to the user is the primary reason that none of the literature's

existing solutions have enjoyed widespread adoption.

As a result, we sought to create a scheme which satisfies the following requirements:

- Total system efficiency is optimized, taking individual processes' parallel efficiencies into account
- No setup, tuning, or manual initialization of any participating application is required before runtime
- No modification to, or recompilation of, any participating application is required
- Effectiveness in both batch and real-time processing scenarios
- System load resulting from both parallel processes which are not truly malleable, as well as processes which are not participating in the SCAF system, is taken into account

1.2 Contributions

Looking at existing research and experiments, the ingredients seem to be available. Some solutions gather information on efficiency by making use of pre-execution profiling [2, 3], while others do not require profiling and do not account for program efficiency [4–7]. However, it is nontrivial to measure and account for program efficiency *without* profiling. This task is made more difficult by the fact that we want to avoid modifying or even recompiling any applications – instrumentation and communication between SCAF processes must be added only as modifications to the

compiler's parallelization runtime libraries.

SCAF solves this problem with a technique which allows a client to estimate its *efficiency* entirely at runtime, alleviating the need for pre-execution profiling. To understand how, consider that parallel efficiency can only be measured by knowing speedup vs serial execution. However, in a parallel program there is no serial equivalent of parallel sections; hence serial measurements are not directly available. Running the entire parallel section in serial is possible, but can greatly slow down execution, overwhelming any benefits from malleability.

We solve the problem of measuring serial performance at low cost by cloning the parallel process into a serial experimental process the first time each parallel section is encountered dynamically. During this first run, the serial experiment is run concurrently with the parallel code as long as the latter executes. Since the serial code is not available in a parallel application, and our goal is to avoid re-compilation, we run the parallel code serialized on one context as the serial code. This is valid since the only relevant speedup for SCAF is versus this serialized parallel code. The parallel process runs on $N-1$ cores, and the serial process on 1 core, where N is the number of threads the parallel process has been allocated. Crucially, the serial thread is not run to completion (which would be expensive in run-time overhead); instead it is run for the same time as the current parallel section. We find this gives a good enough estimate of serial performance to be able to estimate efficiency. *The run-time overhead of the serial experiment is very low because it is run only once during the first dynamic instance of the parallel section. Subsequent dynamic runs of the parallel section run on N cores without running the serial experiment.*

We evaluate the SCAF system in various multiprogramming scenarios and on a wide variety of hardware, showing that SCAF generally improves system efficiency, as measured by sum of speedups for a parallel workload, over both simple equipartitioning and an unmodified OpenMP runtime. Furthermore, we closely examine the major architectural features of our test platforms, and conclude that while NUMA architectures do not seem to be problematic for our scheme, forms of simultaneous hardware multithreading (i.e. SMT) create complex and unpredictable speedup behavior.

We plan to make SCAF open-source by the time of publication. An executable version will be available for users. A source-code version will be available to aid researchers.

Chapter 2

Background

2.1 Multithreading on Unix-like Operating Systems

Threads on Unix-like systems have evolved in terms of implementation, purpose, and standards. To avoid confusion, this section briefly reviews the concept of threads, and clarifies the type of threading implementations targeted by our work.

2.1.1 Threads vs. Processes

While a "process" is an instance of a computer program, a "thread" can be seen as a sort of sub-process working in some way to implement or complete its parent process. In the simplest case, a process is not multi-threaded, and consists of only a single implicit thread. In other cases, a process may request of a threading library that additional threads be created. The work or code executed by these threads must be explicitly provided by the process, either by a low-level programmer or via a higher-level library which has special knowledge of how to task the threads. For example, it is common to write a program which may run on an arbitrary number of threads by employing a high-level language and runtime which knows how to chunk up work known to be independent and distribute it to worker threads. OpenMP is an example of one such language; here, the programmer most commonly asserts to the language that the iterations of certain loops are independent, although other

constructs are available.

Note that the operating system can no more control the number of threads per process than it can the number of processes. There cannot be a paradigm where thread creation is denied for the sake of controlling the degree of parallelism because not all applications use threads to implement parallelism. The process itself determines the number of threads to be used, and how to use them.

2.1.2 Evolution of Threads

It is important to note that the purpose and implementations of software threads have changed over the past few decades. Specifically, before multiprocessor hardware became commonly available, threading implementations were not expected to provide true *parallelism*. These implementations therefore used what is known today as the "N:1" threading model, where all N threads requested by a process were implemented by the operating system as a single entity that would map to only one hardware processor at any point in time [8]. As a result, multithreaded programs implemented on these systems could not benefit from any available parallelism in hardware. This was entirely acceptable on uniprocessor hardware, with no available parallelism. Instead, the purpose of such threading implementations was to allow threads within a common process to be quickly switched between by the threading library without the need to run any kernel code. In a nutshell, the threading libraries implemented concurrency efficiently, without parallelism.

However, with parallel, hardware multithreading processors becoming more

readily available, it became increasingly desirable to implement concurrency *with* parallelism. As a result, almost all major operating systems, including Linux, Solaris, FreeBSD, and OS X now provide threading implementations which abandon the N:1 model in favor of the 1:1 model [8]. The 1:1 model implements each of a process's threads as separate kernel entities which may be scheduled to different hardware contexts simultaneously. As a result, the cost of context-switching is higher, but true parallelism is achieved.

In this work, we expect that all threading implementations are now using the 1:1 model, with threads used for the purpose of increasing parallelism.

2.1.3 Malleable Processes

As threads become more commonly used to take advantage of parallel hardware, many parallel programs and runtimes have been written with the ability to run on parallel hardware with varying degrees of parallelism. In other words, the resulting programs are *malleable*. For example, if a programmer writes a multithreaded image processing program, it will be desirable for that program to be able to run on 1-core, 8-core, 64-core, or even larger machines with minimal effort. Ideally, no code modification or recompilation would be necessary. Parallel languages and runtimes, such as OpenMP, often make effecting this portability quite easy by abstracting the hardware detection and thread creation away from the programmer and user. As a result, a great number of parallel programs are actually malleable. For example, in order to make redistribution easy, all of the OpenMP NAS Parallel Benchmarks [9]

can be run with any number of threads.

2.2 Multiprogramming Strategies

The basic problem of scheduling and allocating for multiple multithreaded processes has a large solution space. While there are many ways to avoid over-subscription, the solutions discussed in this paper largely focus on transforming fine-grained time sharing (the de-facto standard in use today) into space sharing. In this section, we explore several *other* high-level multiprogramming strategies in order to understand why we chose space sharing for SCAF. The set of strategies described here are meant to be representative of the available alternatives, although there are certainly many more possibilities.

In this section, we explore our options in terms of a specific multiprogramming scenario: running NAS benchmarks CG and LU on a 36-core Tilera Tile-GX running Linux. All of the illustrative examples provided are actual recordings of real runs on real hardware.

2.2.1 A Tool for Observing Multiprogramming: `pidpcpu`

To better explore and observe multiprogramming, we have created a new utility called `pidpcpu` and a corresponding `plotpidpcpu` which can be used to plot the results. These tools were born out of necessity: the usual Unix tools (i.e., `top`, `ps`, `mpstat`, etc.) report total CPU time per process tree or thread, but provide little insight into *where* the threads are being scheduled. For example, `top` may reveal that

PID 1234 is using 4 CPUs on average over the last 2 seconds, while PID 5678 used 2. However, any detail beyond this is lost: were the threads involved being moved around to different processors after context switches? Was PID 5678 using 4 CPUs half of the time? `pidpcpu` aims to provide more insight.

`pidpcpu` accepts one or more process IDs as its arguments, and begins tracking where each is run. Because we are interested in observing multithreaded processes, all child processes and threads of the specified PIDs are tracked as one entity. It is potentially expensive to track a large number of threads via conventional methods; therefore, `pidpcpu` uses Linux's recent "CGroups" ("control groups") [10] functionality to obtain accounting information from the specified thread trees. This strategy allows the tool to sample at a relatively high rate (on the order of 100ms) on a live system without significant overhead. Listing 2.1 provides an example of `pidpcpu` invocation and output, with comments provided by the author.

An important feature of `pidpcpu` is that it requires no heavy profiling, and is meant for interactive use. There needn't be any record/analyze cycle. In fact, output from `pidpcpu` can be piped to `plotpidpcpu` for *live* system analysis — similar to `top` or `prstat`, but graphical. `plotpidpcpu` uses `feedgnuplot` [11] and ultimately `gnuplot` [12] as its backend.

2.2.2 Exclusive scheduling

A trivial form of multiprogramming would be *exclusive* scheduling, where the users or runtime strictly avoid multiprogramming by coordinating process execution.

Listing 2.1: `pidpcpu` example on a 4-core Linux system

```
> PERIOD=1 pidpcpu 1111 2222
time  pidtree@cpu  usage
0.0   1111@1       0.99 # PID 1111 on CPU 1...
0.0   1111@2       1.00 # ...and CPU 2.
0.0   1111@3       0.00
0.0   1111@4       0.00
0.0   2222@1       0.00
0.0   2222@2       0.00
0.0   2222@3       0.99 # PID 2222 on CPU 3...
0.0   2222@4       0.98 # ...and CPU 4.

1.0   1111@1       0.99 # 1 second later, no change.
1.0   1111@2       0.99
1.0   1111@3       0.00
1.0   1111@4       0.00
1.0   2222@1       0.00
1.0   2222@2       0.00
1.0   2222@3       0.99
1.0   2222@4       0.98

2.0   1111@1       0.99
2.0   1111@2       0.00
2.0   1111@3       1.00 # Now on CPU 3 rather than CPU 2.
2.0   1111@4       0.00
2.0   2222@1       0.00
2.0   2222@2       0.99 # Switched with PID 1111; now on 2.
2.0   2222@3       0.00
2.0   2222@4       0.98
^C
```

In other words, processes are run one by one. Using `pidpcpu`, let's examine what actually occurs with Linux 3.14 on our 36-core TileGX. Figure 2.1 shows output from `plotpidpcpu` after both processes finish execution, with CG launched first. The horizontal axis of the plot depicts time, while the vertical axis shows CPU time for the shaded process. Note that the vertical space of the graph is segmented per CPU, and essentially 36 vertically stacked plots.

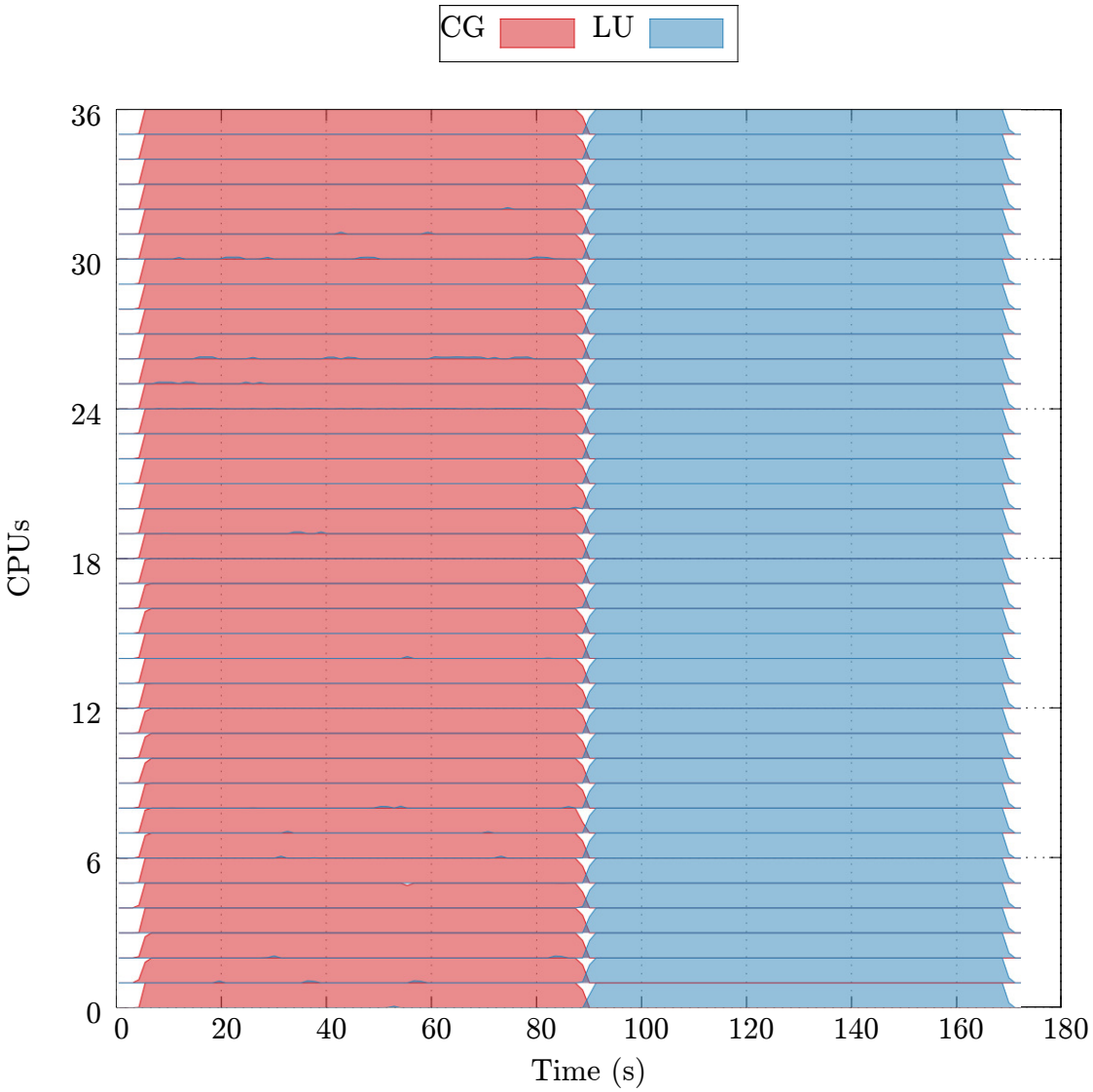


Figure 2.1: Exclusive scheduling on TileGX/Linux

In Figure 2.1, we see that CG begins, and immediately is scheduled across all 36 cores. This is because the OpenMP runtime which CG uses detected 36 hardware cores, and accordingly spawned 36 software threads to load them. During the time period between 0 and 90 seconds, the Linux scheduler sees 36 runnable threads and 36 available hardware contexts; therefore, it schedules a 1-to-1 mapping. This is the scenario that the Linux scheduler and OpenMP runtime anticipate. As a result, each

benchmark runs with minimal context switching and parallel efficiency is reasonably high. In this scenario, the CG benchmark achieves a speedup of 28.9X compared to serial execution, and then LU achieves a speedup of 22.4X compared to serial execution.

2.2.3 Fine-grained multiprogramming

Fine-grained multiprogramming is the de-facto standard, given that it describes what happens when multiprogramming is uncoordinated, and thread scheduling is left solely to the operating system. Figure 2.2 shows output from `plotpidpcpu` when the CG and LU benchmarks are run simultaneously. (E.g., launched from two different shell instances.)

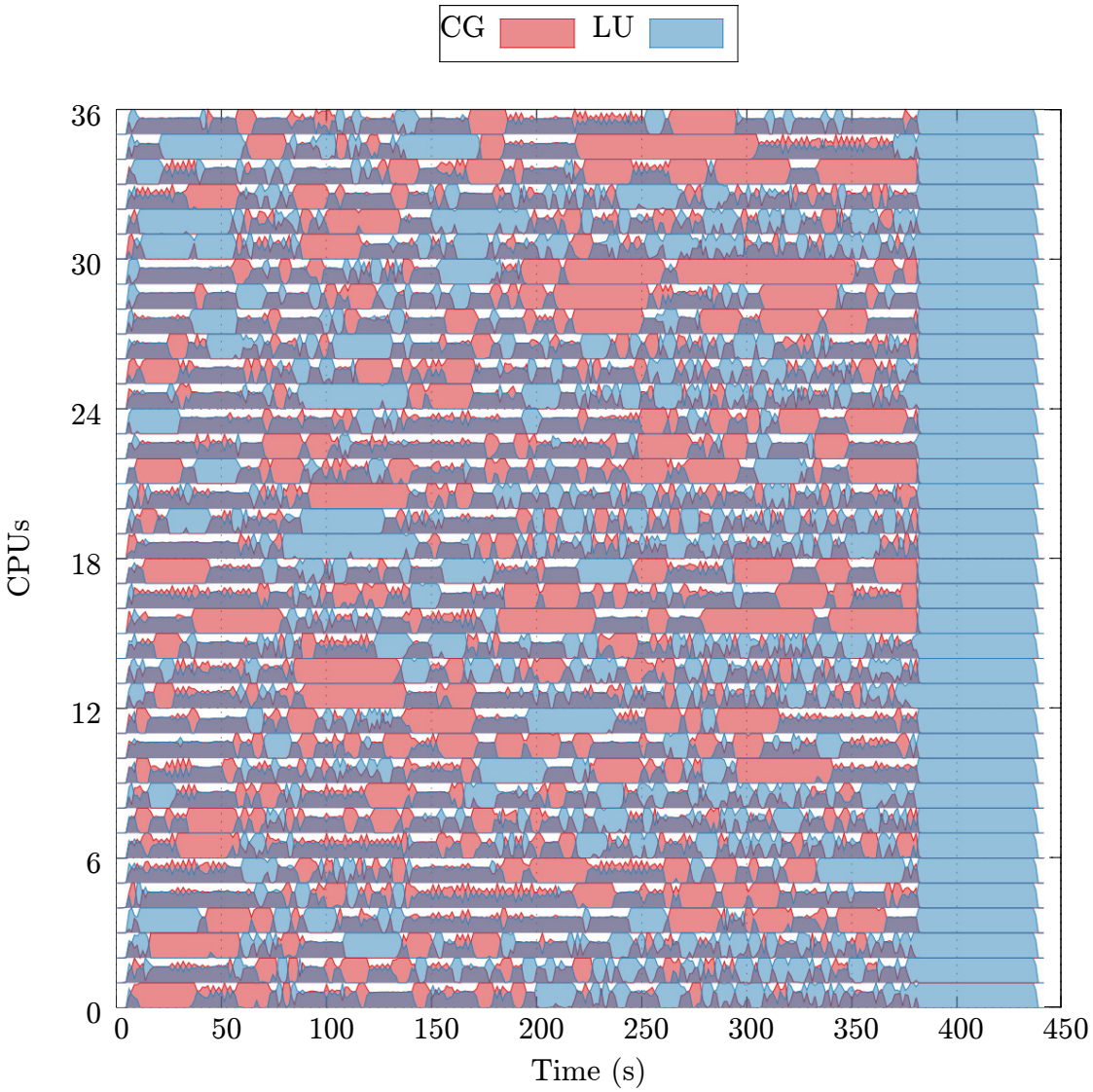


Figure 2.2: Fine-grained multiprogramming on TileGX/Linux

In Figure 2.2, we see CG and LU run concurrently. However, the OpenMP runtimes for both CG and LU each still launch 36 threads, since neither is aware of the other. The result is $36 \times 2 = 72$ runnable threads, which the Linux thread scheduler must map in some way to only 36 cores. A machine in this situation, where there are more runnable threads than available hardware contexts, is "oversubscribed". In order to avoid tracking groups of threads, the Linux scheduler schedules all 72 threads

from the two processes in exactly the same way that it might schedule 72 threads from 72 single-threaded processes. The result is relatively high-speed, fine-grained context switching, as the kernel tries to fairly schedule the 72 threads. Unfortunately, the two groups of threads have dependencies on one another by way of synchronization, which can result in imbalanced progress. Furthermore, the fine-grained context switching of two or more threads on a single hardware context effectively results in threads sharing the hardware. For example, two threads may constantly evict each others' data from a core's local cache, resulting in many cache misses and the need to go to main memory. On both TileGX and x86, LU contains userspace spin-locks and memory fence operations that perform quite badly when not executed on dedicated hardware contexts.

As a result, we see that the parallel efficiency for both processes involved is dramatically lower when the Linux kernel is left to do fine-grained time-sharing. When compared to the processes running alone (exclusively) on the hardware, CG went from 28.9X speedup to 6.4X speedup, and LU from 22.4X to only 4.1X speedup. Note that this is more of a loss in speedup than the 50% one might expect when halving the hardware resources for each process. We argue that this means that the multiprogramming is *not efficient*.

2.2.4 Gang Scheduling

One interesting option which avoids the kernel's fine-grained time sharing is "gang scheduling." With gang scheduling, the threads from a process form a "gang,"

and only one gang runs at a time. While one process is running, the other is essentially paused. In the context of our example, each process forms a gang, and the gangs run for 4 seconds before relinquishing the system to the other process. In this way, the processes are running concurrently, but only 36 threads are runnable at any point in time. In other words, gang scheduling avoids oversubscription.

Figure 2.3 shows CG and LU multiprogrammed via gang-scheduling on the TileGX. Here, we have implemented gang scheduling via the Unix SIGSTOP and SIGCONT signals. Although gang scheduling effects some context switches, it is relatively low-speed. During the multiprogrammed scenario, CG achieves a speedup of 15.6X, which is significantly higher than the 6.4X achieved during oversubscription, and about half of the 28.9X achieved alone. Similarly, LU achieves 10.8X speedup, a marked improvement over 4.1X during oversubscription, and about half of the 22.4X achieved alone. Comparing Figure 2.3 with Figure 2.1, we see that coarse-grained gang scheduling behaves like exclusive (non-multiprogrammed) scheduling but with alternating slices of each process's execution time in order to effect concurrency.

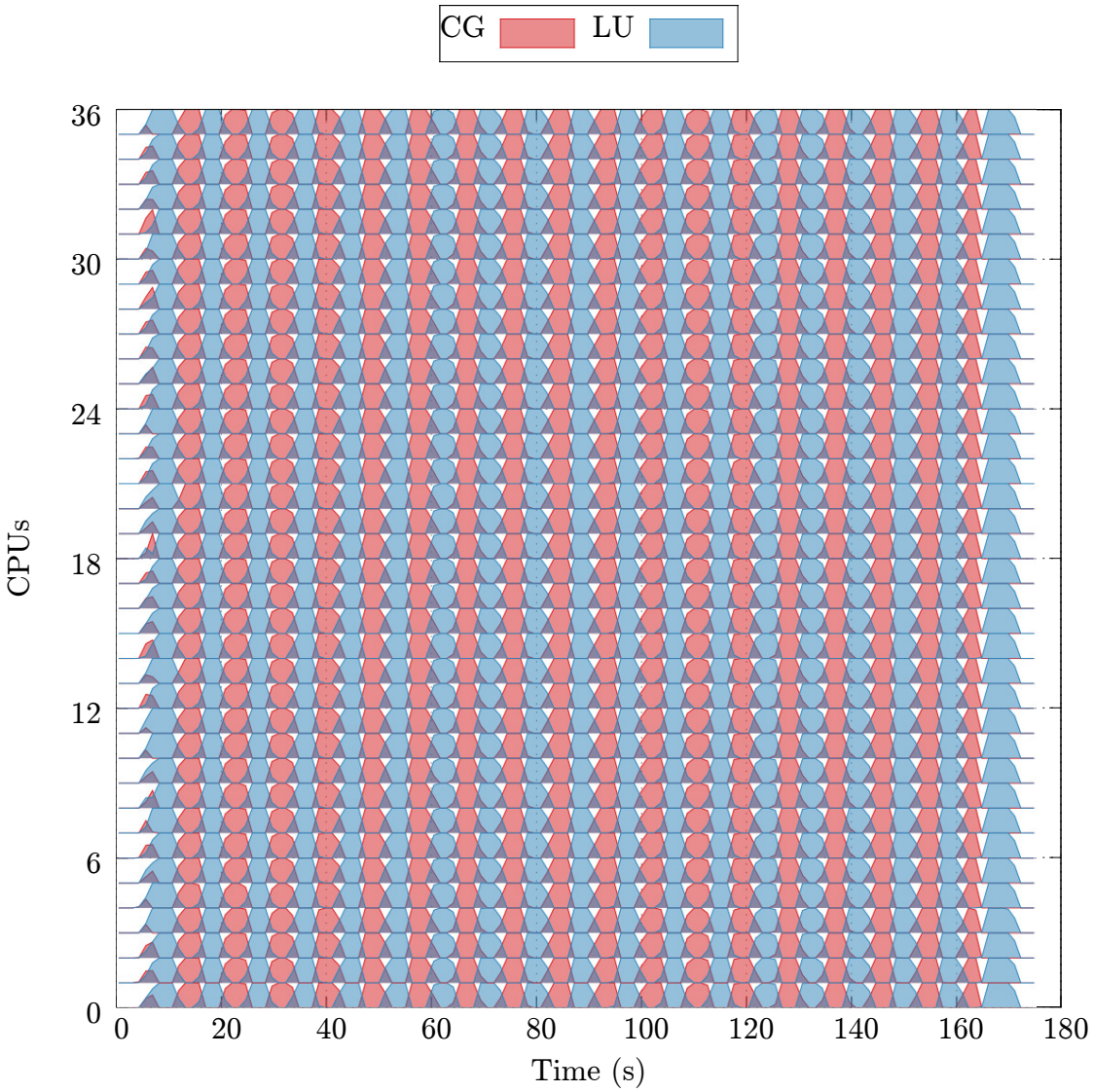


Figure 2.3: Gang-scheduled multiprogramming on TileGX/Linux

We can then see that gang scheduling is an efficient form of multiprogramming: with two processes sharing the machine, we see each slow down by about 50% compared to running exclusively. Unfortunately, the coarse-grained time sharing leaves all but one process completely stopped and unresponsive at any point in time. For this reason, we do not believe that gang scheduling is appropriate for general-purpose use on SMP servers, workstations, and personal computing devices.

Consider a PC with a multi-threaded video player (decoder) in one window, and an unrelated multi-threaded image processor running in another: the video player would effectively pause every 4 seconds! As another example, if a server executing multi-threaded transaction processing such as a web server, an e-commerce site, or a bank site, uses gang scheduling, then the potential transaction delay is increased linearly with the number of gang-scheduled jobs. This would degrade response time to customers to an unacceptable extent. We can of course reduce the period of time which the active gang is runnable, but this will incur more context switches and hardware contention. We argue that gang scheduling is only effective and suitable for particular processing environments and scenarios.

2.2.5 Space Sharing

Given our desire to avoid both fine-grained time sharing and the coarse-grained time sharing of gang scheduling, another attractive option is *space sharing*. With space sharing, the hardware cores split into physical partitions, and processes are constrained to running within those fractions of the hardware. For malleable programs, this can be done by influencing the number of threads that each multithreaded process uses. Fine-grained time sharing and gang scheduling effectively both strive to deal with the condition of oversubcription on the machine, while space sharing avoids oversubscription altogether. The operating system's thread scheduler is then still used, but its job has been made much easier: all it needs to do is map N runnable threads to N hardware contexts, just as with exclusive scheduling or

non-multiprogrammed scenarios.

It is worth noting that both fine-grained time sharing and gang scheduling are solutions which can be implemented within an operating system's thread scheduler. By contrast, space sharing is effectively a higher level form of multiprogramming in that it cannot be performed by the operating system itself. The reason for this is that the operating system does not control the number of threads used by a process: the process itself does. As a result, space sharing must be effected in participation with the part of the processes that manages threads — usually a parallel runtime. In the case of OpenMP programs, the programmer is not in charge of the number of threads used unless they go out of their way to ask the OpenMP runtime for this responsibility. Assuming they do not, the program is then malleable and it is the OpenMP runtime library itself which manages the number of threads used for any given piece of parallel code.

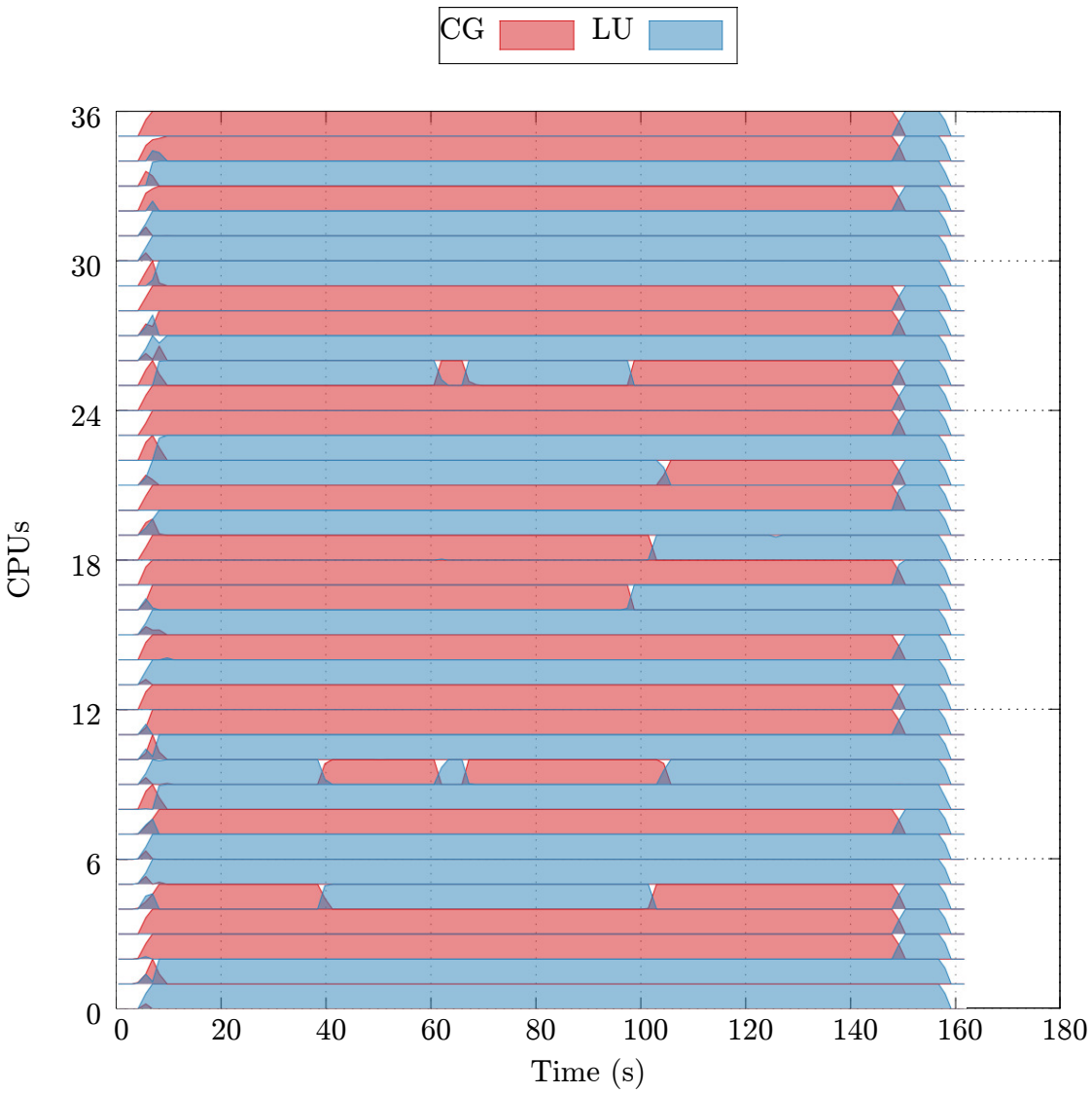


Figure 2.4: Equipartitioned multiprogramming on TileGX/Linux

Figure 2.4 shows CG and LU multiprogrammed via space sharing on our TileGX. The reader may expect to see the top 18 cores' bars running one process, and the bottom 18 cores' bars running the other. However, with space sharing, the regions shared are not necessarily contiguous. The matter of placing threads to specific hardware resources is still left entirely to the operating system's thread scheduler as an orthogonal problem. In fact, Linux now has sophisticated knowledge

of hardware topologies, and attempts to do this mapping intelligently. While space sharing does not deal with placing threads, it does ensure that the total *combined* number of runnable threads is equal to the number of hardware contexts. Scanning the time axis of figure 2.4, we can verify that although threads may migrate between cores, there are exactly 18 threads running each process at any point in time.

With space sharing, we see no high-speed context switching and no contention between software threads for hardware resources. Additionally, both processes are running at all times, ensuring responsiveness as with fine-grained time sharing. At the same time, with space sharing we see that CG achieves 17X speedup and LU achieves 12X speedup. This is an impressive result in terms of efficiency: during exclusive scheduling, the machine achieved either 28.9X speedup or 22.4X speedup, depending on which process was running. With space sharing, at any point in time during the multiprogrammed region, the machine was achieving about $17 + 12 = 29$X speedup. This represents an improvement in terms of efficiency for the machine's 36 cores. It's not hard to see how this is possible: space sharing works by reducing the number of threads and therefore the degree of parallelism used by each process, and in general parallel processes are more efficient at lower degrees of parallelism. (Specifically, in the common case where parallel efficiency looks like $\frac{1}{(1-par+\frac{par}{p})\cdot p}$, where par is the fraction of runtime that is parallel and p is the number of threads, efficiency only decreases as p increases.)

Technique	CG Speedup	LU Speedup	Advantages	Disadvantages
Exclusive scheduling	28.9X	22.4X	Avoids all concurrency	Must wait to start LU
Fine-grained time sharing	6.4X	4.1X	Allows concurrency	Low speedups from contention
Gang scheduling	15.6X	10.8X	Avoids contention	Impacts latency
Space sharing	17.0X	12.0X	High efficiency, latency unaffected	Requires thread control

Table 2.1: Summary of multiprogramming techniques in CG-LU scenario

Technique	Shared Dimension	Concurrency	Responsiveness	Oversub-scription
Exclusive scheduling	time	no	yes	no
Fine-grained time sharing	time	yes	yes	yes
Gang scheduling	time	yes	degraded	no
Space sharing	space	yes	yes	no

Table 2.2: Summary of multiprogramming techniques

2.2.6 Discussion

Table 2.1 summarizes the four multiprogramming techniques described in this section. Exclusive scheduling is examined mostly as a baseline: it does not achieve multiprogramming, but demonstrates the scalability of CG and LU while running alone on the TileGX. Fine-grained time sharing, which is what is used by default on most major multi-threaded operating systems that we're aware of, correctly achieves concurrency, but exposes too much parallelism to the operating system's thread scheduler. Furthermore, LU and the OpenMP runtime library itself utilize synchronization strategies which perform poorly when threads must share hardware contexts. It is possible to avoid these strategies (e.g., spin-locks) in general to avoid this contention, but changing synchronization mechanisms may incur its own performance penalty. Gang scheduling is another interesting technique which guarantees that each process runs on the entire machine, and therefore that each thread runs on its own hardware context. In our simple example, the performance results from gang scheduling are good, and a rough sort of concurrency is achieved. However, this is accomplished via a sort of coarse-grained time sharing which impacts latency. In particular, we can imagine this being unacceptable for transaction processing and interactive applications. Space sharing avoids this problem by controlling the number of threads used, and therefore the fraction of the machine used by each process. This avoids contention, maintains responsiveness, and scales well for for multiprogramming scenarios with more than two processes. The disadvantage to space sharing is that it requires cooperation from processes in some way. They must be malleable, or able to

control the number of runnable threads. Fortunately, because parallel applications are already developed to be portable to machines with varying numbers of hardware contexts, malleable processes are not uncommon. As a result, we focus on facilitating multiprogramming specifically by way of space sharing malleable programs in our efforts.

Chapter 3

Related Work

SCAF seeks to solve performance and administrative problems related to the execution of multiple multithreaded applications on a many- or multi-core shared-memory system. This section outlines related work, both in the world of shared-memory parallelism, and in the world of distributed-memory parallelism. Table 3.1 contains a summary of the features of related work, and compares them to SCAF. All of the methods enumerated in the table are discussed in more detail in this section.

Implementation	Avoids recompilation	Avoids modifications	Considers process efficiency	Avoids a priori testing / setup	Multi-process support
SCAF	✓	✓	✓	✓	✓
RSM [4]	×	×	×	✓	✓
DTiO [5]	✓	✓	×	(N/A)	✓
ERMfMA [6]	×	×	×	(N/A)	✓
APiCPC [2]	×	✓	✓	×	✓
Hood [13]	×	×	×	(N/A)	✓
PCfMSMP [7]	✓	✓	×	(N/A)	✓
Lithe [14]	✓	✓	×	(N/A)	×
CDPAS [3]	×	×	✓	×	✓

Table 3.1: Feature comparison of related implementations (ad hoc acronyms used for brevity)

3.1 Distributed Memory Systems

Flexible and dynamic scheduling for distributed memory parallel applications and systems has been an active area of research. SCAF does not compete in the distributed memory world, as it is designed to solve problems pertaining to shared-memory systems. In particular, work in the distributed memory domain tends to focus on mechanisms allowing applications to dynamically migrate and adapt [15–25], or on scheduling for batch-type systems [1,15,26–37]. On single shared-memory systems, migration is not an issue and batch scheduling is rarely used. However, since the problems of distributed systems are similar at a high level, we briefly describe some of the related work in this section.

Kale et al [38] implemented a system for dynamically reconfiguring MPI-based applications through a system using a processor virtualization layer. Crucially, this allows the migration of work from one node of the distributed system to another. Load balancing is effectively achieved by creating many virtual processes for each physical processor. The system then can reconfigure parallel jobs at runtime based on the arrival or departure of other jobs. However, recompilation of a participating application is required, and small modifications to the source code are necessary.

Sudarsan et al [39] improved on this work with ReSHAPE, their framework for dynamic resizing and scheduling. Using the provided resizing library and API, application users can specify shared variables suitable for redistribution between iterations of an outer loop. The points at which redistribution is safe must be specified by the programmer. Between each iteration, a runtime scheduler makes decisions on

whether to expand or shrink a job based on node availability and observed whole-application performance. The primary disadvantage of ReSHAPE is that it requires applications to be significantly rewritten to use their API.

3.2 Shared Memory Systems

Relatively little work has been done concerning multiprogrammed, multithreaded scheduling and the problem of oversubscription. Tucker et al [7] observed serious performance degradation in the face of oversubscription on shared-memory multiprocessors. They showed that by modifying a version of the Brown University Threads package used on an Encore Multimax, a centralized daemon can (strictly) limit the number of running threads on the system to avoid oversubscription by suspending threads when necessary. By modifying only the system's threads package, they were able to support many programs using that threads package without modification. However, their work has several disadvantages as compared to SCAF work: (1) the partitioning policy does not take into account any runtime performance measurements, but assumes all processes are scaling equally well, and (2), the scheme's ability to control the running number of threads depends on the use of the specific parallel paradigm where the programmer creates a queue of tasks to be executed by the threads, and the assumption that the application does not depend on having a certain number of threads running. If an application does not meet both requirements, then it may run incorrectly without warning. This is a restriction of operating within a threads package where unsupported program behavior cannot always be

detected at runtime. By contrast, SCAF offers modified runtime libraries which provide higher-level abstractions. Unsupported program behavior which would imply non-malleability is detected as it is requested, after which SCAF avoids incorrect behavior by holding the number of threads fixed for that running program.

Arora et al [40] designed a strictly user-level, work-stealing thread scheduler which was implemented in the "Hood" [13] prototype C++ threads library, and later in the Cilk-5 [41] language's runtime system. Work stealing is an approach in which the programmer specifies all available parallelism in a declarative manner, and then the implementation schedules parallel work to a certain number of worker threads (using deques, or simply work stealing queues) which are allowed to "steal" work from one another in order to load balance. The number of worker threads is usually equal to the number of available hardware contexts. The problem that Arora solves is that when multiple processes are running, each doing work stealing with multithreading, then having independent worker threads for each process leads to more worker threads than hardware contexts, leading to over-subscription of the machine and poor performance. Their approach is to combine the work-stealing queues across applications, and using a number of shared workers that does not result in oversubscription. Their approach also accounts for serial processes when avoiding oversubscription.

The approach used in Hood [13] has several differences with the goals and capabilities of SCAF. First, Hood can only reduce oversubscription when the parallel processes all utilize work-stealing libraries. In contrast, SCAF reduces oversubscription for any malleable parallel processes, regardless of whether they use work

stealing or not. Second, although Hood and SCAF have the same goal of avoiding oversubscription, they do so using different mechanisms: SCAF relies on malleable processes to reduce the number of threads, whereas Hood has a specialized solution for work stealing that relies on the work stealing programming model, without taking advantage of malleability. Third, parallel threads using Hood are not allowed to use blocking synchronization, since Hood might swap out a lock-holding thread, which would prevent other threads from making progress. SCAF has no such restriction, since it reduces the number of threads created, rather than allowing threads to be swapped out. Fourth, the implementation of Hood is complex and difficult since it has the same restriction as user processes, in that Hood code must not use blocking synchronization, which is simpler, but might cause tremendous slowdowns if the kernel preempts a process which holds locks, causing other workers to block. Fifth, Hood does not take any run-time measurements, and hence cannot favor processes with better scalability, whereas SCAF does, which helps to improve overall system throughput by rewarding processes that scale better.

Hall et al [2] performed experiments that emulate using a similar centralized daemon and modifications to the Stanford SUIF auto-parallelizing compiler to dynamically increase or decrease the number of threads at the start of a parallel section based on system load and runtime measurements of how effectively each parallel section uses its hardware contexts. Kazi et al [3] adapted four parallel Java applications to their own parallelization model and implementation so that each application reacts to observed system load and runtime performance measurements in order to increase or decrease its number of threads at runtime before each parallel section.

SCAF builds on ideas developed in these works. Compared to SCAF, their systems have the following drawbacks: (1) they require recompilation or modification of the programs in order to control the number of threads; and (2) despite controlling compilation, they are unable to avoid depending on a priori profiling for making allocation decisions. SCAF works with unmodified, SCAF-oblivious binaries, and collects all of its information regarding efficiency during program execution, avoiding the need for careful application profiling.

Suleman et al [42] describes a feedback-based system for choosing the optimal number of threads for a single program at runtime. Specifically, the system can decrease the number of threads used in order to improve efficiency in the face of critical sections and bus saturation. This system requires no a priori knowledge of programs, and utilizes a serialized "training" phase to reason about serial and parallel performance. However, the system does not attempt nor claim to reason about multiprogramming, and it is unclear if it could be adapted to do so. SCAF carefully avoids any serialization and seeks primarily to handle multiprogramming. Other related works use varying techniques and metrics to control single-program parallelism [43–51], but similarly do not explore multiprogramming.

More recently, Pan et al [14] created the "Lithe" system for preventing hardware oversubscription within a single application, or process, which composes multiple parallel libraries. This is a separate problem from the one discussed in this paper. For example, consider a single OpenMP-parallelized application which makes a call to an Intel TBB-parallelized library function. The result is often significant oversubscription: on a system with N hardware contexts, the OpenMP parallel section will

allocate N threads, and then each of those threads will create another N threads when Intel TBB is invoked, resulting in N^2 threads. The Lithe system transparently supports this composition in existing OpenMP and/or Intel TBB binaries by providing a set of Lithe-aware dynamically-loaded shared libraries. However, it should be made clear that Lithe makes no attempt to coordinate multiple *applications* running concurrently, and does not vary the number of threads which the application is using at runtime. Like much of the work it improves upon [52–55], Lithe strictly avoids oversubscription potentially resulting from composition of parallel libraries within a single process. SCAF builds on Lithe's idea of supporting existing applications via modified runtime libraries, but focuses instead on the composition of parallel libraries used in separate, concurrently-executing executables.

McFarland [4] created a prototype system called "RSM," which includes a programming API and accompanying runtime system for OpenMP applications. The application must be modified to communicate with the runtime system via API calls between parallel sections. Once recompiled, the application communicates with the RSM daemon and depends upon it for decisions regarding the number of threads to load beginning with the next parallel section. The RSM daemon attempts to make allocation decisions according to observations of how much work is being performed by each process at runtime. An application's useful work is taken to be the number of instructions retired per thread-seconds. Processes are given larger allocations if they perform more useful work. Unlike RSM, SCAF does not require program recompilation. Further, SCAF compares *efficiency* observed at runtime,

considering the *improvement* in IPC[1] gained by parallelization, whereas RSM only considers the *absolute* IPC of each process.

In the interest of preserving existing standards and interfaces, Schonherr et al [5] modified GCC's implementation of OpenMP in order to prevent oversubscription. The implementation supports applications without recompilation. However, their system implements only a simple "fair" allocation policy, where all applications are assumed to scale equally well, and no runtime performance information is taken into account.

Hungershöfer et al [6] implements a runtime system and daemon for avoiding oversubscription in SMP-parallel applications. Their system requires modifications to the applications involved, and provides a centralized server process which controls thread allocation. However, their method for maximizing accumulated speedup depends on significant offline analysis of the applications for determining their speedup behaviors, parallel runtime components, and management/communication overheads.

3.3 Related Implementations

As part of related work, some solutions have been implemented and explored. These are listed below in order of their similarity to SCAF, and their features are enumerated in Table 3.1.

1. RSM, [4]

2. Dynamic Teams in OpenMP, [5]

[1]Instructions per cycle

3. Efficient Resource Management for Malleable Applications, [6]

4. Adapting parallelism in compiler-parallelized code, [2]

5. Hood, [13]

6. Process control and scheduling issues for multiprogrammed shared-memory processors, [7]

7. Lithe, [14]

8. A comprehensive dynamic processor allocation scheme for multiprogrammed multiprocessor systems, [3]

Chapter 4

Design

4.1 System Overview

A system running SCAF consists of any number of malleable processes, any number of non-malleable processes, and the central SCAF daemon. The SCAF daemon is started once, and one instance serves all users on the system. All processes are SCAF-oblivious, and are started by users in the usual uncoordinated fashion. Parallel binaries load SCAF-compliant runtime libraries in place of the unmodified runtime libraries — this does not require program modification or recompilation. The SCAF-compliant libraries automatically determine whether a process is malleable at runtime, transparently to the user. A non-malleable process will proceed as usual, requiring no communication with the SCAF daemon, while a malleable process will consult with the SCAF daemon throughout its execution. A process which loads no SCAF-compliant parallel runtime libraries is assumed to be non-malleable and proceeds normally. The SCAF daemon is responsible for accounting for the load incurred by any non-malleable processes. Specifically, hardware contexts used by non-malleable processes must be considered unavailable to malleable processes.

4.2 Conversion from Time-sharing to Space-sharing

By default, modern multi-user operating systems support the execution of multiple multithreaded applications by simple time-sharing. Parallel processes are unaware of one another, and each assume that the entire set of hardware contexts is available. In general, this results in poor efficiency and performance unless the system is otherwise quiescent (*i.e.*, unloaded). Refer back to Section 2.2.3 for more details on this behavior. As an extreme example, we found that on a small 4-core Intel i5 2500k system running Linux 3.0, the per-instance slowdown when running two instances of the NAS NPB "LU" benchmark (each on 4 threads) was as much as a factor of 8 when compared to using space sharing. With the same hardware running FreeBSD 9.0 the penalty was much greater, exhibiting a slowdown by a factor of more than 100. An investigation revealed that LU implements spinlocks in userland which perform poorly without dedicated hardware contexts [56]. Modifying the synchronization primitives used by the system's `libgomp` runtime library won't help, since the problematic synchronization lies in LU itself. Short of perhaps modifying the application, the best solution is space sharing.

The objective of the SCAF system is essentially to effect space-sharing among all hardware contexts running on the system, such that the operating system can schedule active threads to idle hardware contexts and avoid the fine-grained context-switching and load imbalances incurred by heavy time-sharing.

4.3 Sharing Policies

In order to justify the policies which the SCAF daemon implements, a brief discussion of possible policies is useful. The following terminology is used:

- Runtime of a process j: T_j
- Speedup of a process j: S_j
- Threads allocated to process j: p_j
- Number of hardware contexts available: N
- Number of processes running: k

Additionally, we define per-process "efficiency" as $E \equiv \frac{S_j}{p_j}$.

4.3.1 Minimizing the "Make Span"

In distributed memory systems, where users generally submit explicit jobs to a space-sharing job manager, the de facto goal is to minimize the "make span," which is the amount of time required to complete all jobs.

However, the algorithms to solve this problem require precise information concerning not only the speedup behavior of each job, but also accurate estimates of the total work required until a job's completion. This implies a batch-processing model, possible for large distributed memory machines. On shared-memory systems, jobs are run without prior intimation by the user, so the run-time system cannot predict when applications will start, nor when a running application will end. As a result, the make span cannot be applied. Multithreaded processes which operate

on a virtually infinite stream of input data or requests are also not uncommon. In these cases, the "make span" cannot be applied since processes do not necessarily terminate. This actually represents a large class of applications including web servers, web caching software, file servers, transaction processing, database servers, customer relationship management software, and many more.

Therefore, a new goal is required for a runtime system such as SCAF. Given that the future system load cannot be predicted by the run-time system, we seek an instantaneous metric which will allow SCAF to reason about the performance of processes at runtime. Furthermore, the optimization problem should be constrained such that the system's behavior is consistent with the expectations of an interactive shared-memory machine.

4.3.2 Equipartitioning

When performing equipartitioning, fully "fair" sharing of the hardware resources is achieved, without concern for how efficiently said resources are being used. Each process occupies an equal number of hardware contexts:

$$p_j \leftarrow \left\lfloor \frac{N}{k} \right\rfloor \tag{4.1}$$

The remaining ($N \bmod k$) hardware contexts are distributed arbitrarily among ($N \bmod k$) processes to ensure full utilization.

The clear advantage to equipartitioning is simplicity. In fact, with equipartitioning, there is no optimization problem, and no performance metrics need be considered: it is merely a policy. Oversubscription and underutilization are avoided,

and no a-priori performance measurements are required. The problem with equipartitioning is that it can result in low system efficiency. For example, given program A with $S_A(p_A) = 1 + \frac{4}{5}(p_A - 1)$ and program B with $S_B(p_B) = 1 + \frac{1}{8}(p_B - 1)$, one might intuitively want to allow the better-behaved program, A, to use more hardware contexts than B since it makes better use of each hardware context. However, equipartitioning ignores observed speedup. The number of threads that each process receives is simply fixed, with no optimization goal in mind. With SCAF, we seek to improve on the equipartitioning allocation policy by rewarding processes with more threads when they are observed to scale well with more threads.

4.3.3 Maximizing System IPC

In order to improve upon equipartitioning and make smarter allocations, a tempting approach is to maximize system-wide IPC (Instructions Per Cycle). However, in this subsection we will see that this optimization goal leads to a poor allocation policy.

IPC is an attractive metric for processes because it can actually be sampled as an instantaneous metric for an individual process, including all threads. (See Section 2.1.1 for a discussion of threads and processes.) Perhaps equally important is the fact that IPC is already a familiar performance metric for computer architects and software developers.

Consider an allocation policy, "max-IPC," for space sharing which allocates p_j threads to process j such that the sum of IPC achieved over all N hardware contexts

is maximized. In the context of the TileGX example examined in Section 2.2, what would this look like? Figure 4.1 shows output from `pidpcpu` and `plotpidpcpu` (see Section 2.2.1), showing the behavior of CG and LU when space-shared optimally for system IPC. For comparison, refer back to Figure 2.4, which shows CG and LU space-shared according to equipartitioning.

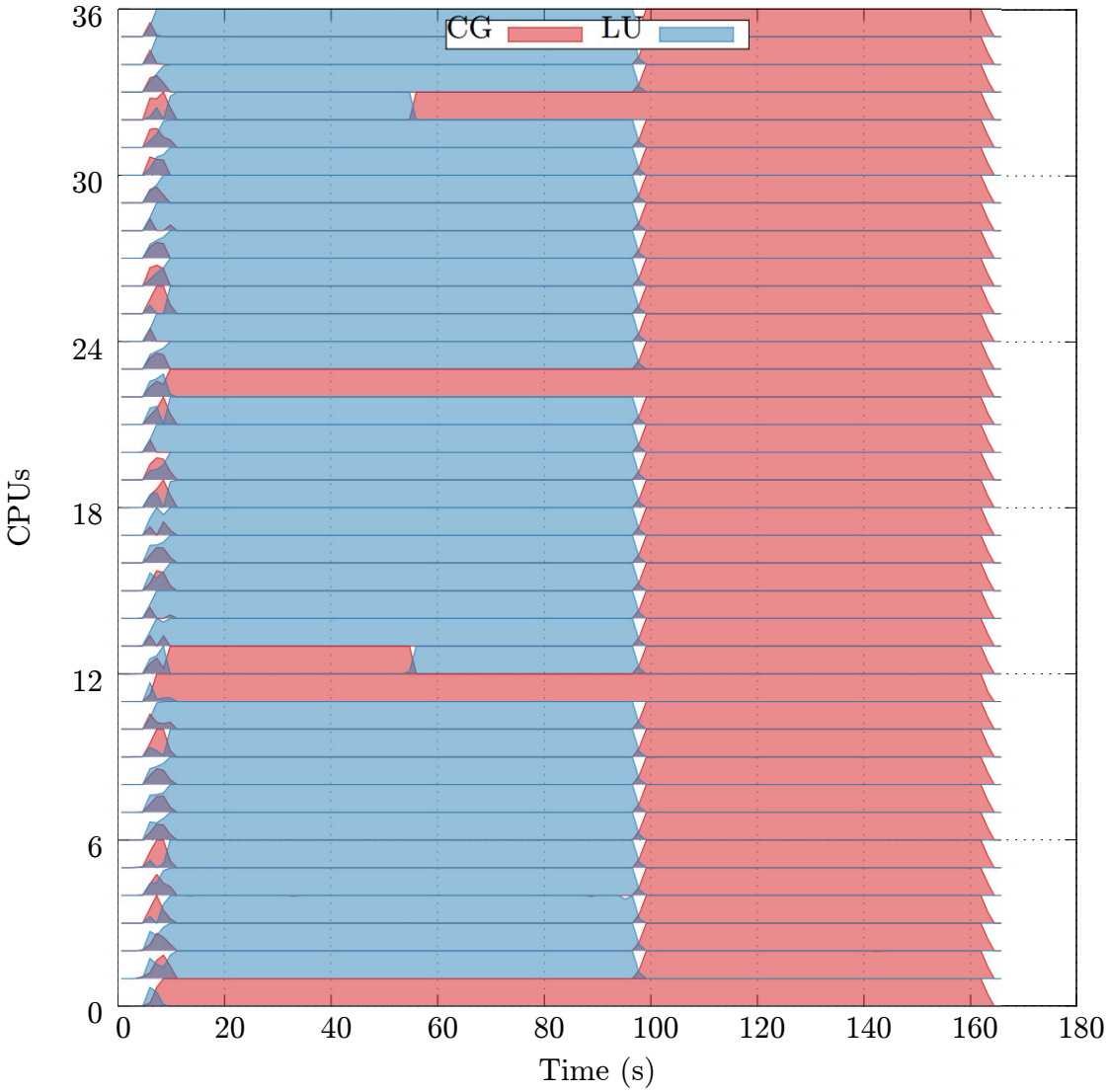


Figure 4.1: Space sharing allocated for maximum system IPC on TileGX/Linux

In Figure 2.4, during the multiprogrammed section (time 0 to 100 seconds) CG

is achieving 6.1 IPC, while LU is achieving 11.7 IPC. After time 100 seconds, LU finishes its execution and multiprogramming ends.

The important thing to note about Figure 4.1 is that while both processes are running, the LU process has been given all threads except four. Why is this? The allocations were chosen by our max-IPC policy at all points in time to maximize the system's total achieved IPC, and it turns out that the individual threads in LU have a much higher IPC than those in CG. As a result, the strategy for max-IPC in this case is simple: give LU *all* of the threads. Since assigning CG 0 threads would devolve our system into exclusive scheduling, we instead allocate CG 4 cores and LU the other 32. This strategy does indeed improve IPC: with equipartitioning, CG achieved 6.1 IPC on 18 cores, while LU achieved 11.7 IPC on the other 18 cores; with max-IPC, CG achieved only 1.2 IPC, while LU achieved a whopping 22.4 IPC. Just the 22.4 IPC achieved by LU alone is now significantly greater than the $6.1 + 11.7 = 17.8$ IPC seen with equipartitioning.

We have seen that system IPC has been improved, and the author asserts that this is indeed the configuration which maximizes IPC in this example. We are left with the impression that by maximizing IPC, we have in some sense maximized the utilization or efficiency of the machine. Unfortunately, we will see that this is untrue. Did maximizing IPC actually make better use of the hardware? To judge this, we'll refer to Table 4.1, which summarizes the changes seen in our CG+LU scenario when moving from equipartitioning to max-IPC.

Crucially, Table 4.1, includes high-level *efficiency* and *speedup* numbers, only available after each process finishes execution. We see while system IPC has improved

from 17.8 to 23.6 IPC by increasing LU's allocation, total system efficiency (i.e., the average speedup contributed by each core) has decreased from 0.81 to 0.67! In fact, giving LU more cores was exactly the wrong move: it turns out that CG is easily the more efficient parallel process, and therefore would have made better use of additional cores.

Max-IPC favored LU merely because it had a higher IPC than CG. However, in general, *two IPC metrics are only meaningfully comparable when the instructions ("I") are the same.* This is due to the fact that some instructions require more or less time to execute, depending on the hardware architecture. In the field of computer architecture, a higher IPC is often used as evidence of faster hardware, with the assumption that the workload binary remains constant. By favoring processes with threads running at higher IPC rates, the optimization goal of Max-IPC is implicitly doing exactly the opposite: concluding that one binary's execution is outperforming another's on the same hardware based on its higher achieved IPC. This comparison is essentially meaningless, since we cannot assume that our processes consist of similar streams of instructions.

For example, a floating square root instruction on a given machine may require 100 cycles, while an integer addition instruction may only require 2 cycles. As a result, a square root-intensive parallel process will likely exhibit a very low IPC per speedup (say, 0.01), while an integer addition-intensive parallel process may exhibit very high IPC per speedup (say, 1.0). Now, say both processes are implemented with perfect parallel efficiency. Max-IPC would dictate simply giving the addition-intensive parallel process the entire machine, simply because it uses shorter instructions! This

Technique	Process	Threads	Total IPC	Speedup	Efficiency
EQ	CG	18	6.1	17X	0.94
	LU	18	11.7	12X	0.67
	Sum/system	36	17.8	29X	0.81
Max-IPC	CG	4	1.2	4X	1.00
	LU	32	22.4	20X	0.63
	Sum/system	36	23.6	24X	0.67

Table 4.1: Results of Max-IPC compared to Equipartitioning

is plainly undesirable: the higher per-thread IPC of the integer workload is merely an artifact of the instructions used, with no bearing on parallel or even architectural efficiency. We argue that maximizing system IPC is not a meaningful goal: it does not directly take scalability into account, and in many cases will simply unfairly allocate in favor of processes with denser sequences. We are certainly not the first to argue that IPC is likely misleading when naively used to analyze multi-threaded behavior: Alameldeen and Wood [57] arrives at much the same conclusion.

4.3.4 Maximizing the Sum Speedup

Another appealing goal is to maximize the total sum of speedups achieved by the running processes. That is, given a function $S_j(p_j)$ describing the speedup of each process with p_j threads, maximize $\sum_{\forall j} S_j(p_j)$ by choosing p_j for all $j = 1 \ldots k$. By maximizing the sum speedups, the average speedup obtained per process is maximized. If allocations such as p_j are fixed throughout a program's execution, then this optimization problem only needs to be evaluated one time, when the processes begin execution.

Maximizing the sum of speedups achieved makes sense in the context of choosing core allocations: the ideal outcome of allocating a core is that it will contribute a large improvement to the relevant process's performance. This is exactly "speedup." Consider a theoretical "Max-speedup" allocation policy, which has perfect knowledge of all process's current and near-future speedup behavior. In other words, it knows $S_j(p_j)$ for all j, all current conditions, and for all points in time of process execution. At a given instant in time, to maximize speedup, Max-speedup can follow a simple

algorithm to maximize total speedup:

Algorithm 1: Max-speedup, given perfect knowledge of speedup behaviors

```
N_available ⟵ N
for j ⟵ 1 to k do
    p_j ⟵ 1
    N_available ⟵ N_available − 1
while N_available ≥ 1 do
    improvement_max ⟵ 0
    chosenProcess ⟵ NONE
    for j ⟵ 1 to k do
        improvement ⟵ S_j(p_j + 1) − S_j(p_j)
        if improvement > improvement_max then
            improvement_max ⟵ improvement
            chosenProcess ⟵ j
    p_chosenProcess ⟵ p_chosenProcess + 1
    N_available ⟵ N_available − 1
```

Algorithm 1 will result in ideal allocations p, but fatally depends on a fully defined and accurate S_j for all $j \in k$. Discovering and describing S becomes a complex problem with malleable processes where both the speedup function S_p and process allocations can effectively change over time. For example, different parallel sections of code in the same program may vary in how well they make use of hardware

contexts. Even if we perform extensive testing and characterization of each parallel section in applications before runtime, in general parallel efficiency may still vary unpredictably due to inputs to the processes. Not only would fully describing S be enormously complex, it would also expensive in terms of testing time, and a significant setback for users. We seek to implement a system which is as easy to use as current systems, without requiring that all of the applications involved be heavily profiled and deeply understood for all possible inputs. Therefore, what is needed is a system which simplifies the problem slightly, such that efficiency observed only at runtime is taken into account, and optimization of allocations is made with only the near future in mind.

4.3.5 Maximizing the Sum Speedup Based on Runtime Feedback

This is the approach we have devised which is used in SCAF. The goal is to partition the available hardware contexts to processes quickly, adjusting over time according to information available at runtime. Rather than optimizing for the entire duration of process runtimes, which cannot be known, we optimize only for the near future. However, the details of such a system are not immediately clear. When should allocation decisions be made? How do we reason about speedups?

One can begin to imagine a system in which allocation decisions are made per parallel region. However, these parallel regions often begin and end execution at a very high frequency. Hence changing the thread allocation for each parallel region is infeasible since the costs of thread initialization and termination as well as

allocation computation would result in prohibitively high overhead. Ideally, the allocation should change relatively infrequently and asynchronously. However it *should* change after longer intervals during an application's run-time, since the application's behavior may change over time, perhaps because it moves to a different phase in the execution. As a corollary, since we cannot possibly react to individual parallel regions, we should reason about speedups in a per-process manner. This also has the advantage of keeping the scheme adaptable to parallel processes that do not use the paradigm of parallel regions.

Consequent to the discussion above, SCAF clients must maintain and report a single efficiency estimate per process. It is the client's responsibility to distill its efficiency information down to this single constant, and refine it over time. This is a nontrivial task for a pure runtime system since capturing efficiency information requires information on the serial performance of sections. SCAF's lightweight serial experiments, discussed in section 5.2.1, represent a solution for gaining this information without incurring the penalty of temporary serialization, which can be extremely expensive.

This strategy for estimating efficiency is best explained by referring back to our CG+LU example. Starting with equipartitioning, we would like to be able to reason somehow that CG will scale better than LU, and allocate more threads to CG. However, at run time we cannot resort to temporarily serializing a parallel section for timing purposes, since it may run for an unknown period of time. Therefore, the "speedup" and "efficiency" columns in this table are unavailable at run time. (Gray shading is used to indicate this.) We have IPC rates for each parallel process, but

Process	Threads	Total IPC	Serial IPC	IPC Ratio		Speedup	Efficiency
CG	18	6.1	0.35	6.1/0.35 =	**17.4**	**17**X	0.94
LU	18	11.7	0.85	11.7/0.85 =	**13.7**	**12**X	0.67

Table 4.2: Using IPC ratios to estimate speedup for runtime feedback

they are not comparable: CG speeds up better than LU yet has a lower IPC. However, if we can somehow obtain an idea of the *serial* IPC of each process, then we find that the ratio of parallel IPC to serial IPC provides a reasonable approximation of the unavailable speedup. In the case of CG and LU, we see that CG's lower parallel IPC explained by its lower serial IPC. More importantly, we see that CG nets a greater relative IPC improvement than LU when run in parallel. With serial experiments providing a serial IPC estimate, we can usefully compare these improvements and attempt to reason about speedup and efficiency. Without serial experiments, we have no context for IPC measurements and thus no basis for comparison.

To support our claim that using the ratio of parallel IPC to serial IPC for a program can usefully approximate speedup, please refer to Figures A.6-A.14 in the Appendix, which compare the IPC ratio (plotting with points) to actual speedup for nine NAS benchmarks on TileGX. Although the results here are very accurate, we do not claim that IPC ratio is a perfect indicator of speedup in general. When cycles counts include instructions which are not useful work (such as barriers), this estimate can be misleading. Our hope is that SCAF clients can avoid counting these cycles since many of these synchronization constructs will be in the modified runtimes (e.g., `libgomp`) themselves.

By default, lightweight experiments return instructions per cycle (IPC) as measured by the PAPI runtime library [58] since this is generally available from hardware counters. PAPI returns the IPC achieved by the calling thread alone. However, it's worth noting that experiments could return any metric which is generally indicative of program progress. For example, floating point operations completed per second (also available from PAPI) may be a more reliable indicator of work if, for example, it is known that the machine is primarily used for floating point work. This is a simple compile-time option in SCAF. However, while it is well understood that IPC is not an ideal work performance metric, we choose to use it by default in SCAF in order to avoid limiting SCAF's usefulness to just floating point workloads.

From a lightweight experiment, SCAF obtains an estimate of the serial IPC of a section. This measurement is then used later at runtime to compare against observed parallel IPC measurements in order to estimate the efficiency for that specific parallel section and the process.

The efficiency estimate allows the central SCAF daemon to reason about how efficiently each process makes use of more cores relative to the other clients (processes). Specifically, the daemon uses this efficiency estimate to build a simple speedup model

$$S_j(p_j) \approx 1 + C_j \log p_j, \text{ where } C_j \leftarrow \frac{E_j p'_j - 1}{\log p'_j} \tag{4.2}$$

where E_j is the reported parallel efficiency from client j, and p'_j is the previous allocation for j. This can be thought of as the simplest form of curve fitting, where the

only parameter to the curve is the constant factor C_j. The model describes a simple sublinear speedup curve specified by tuples (number of threads, speedup) which goes through the points $(1, 1)$ and $(p'_j, E_j p'_j)$, since $S_j = E_j p'_j$. More sophisticated speedup models have certainly been developed [59, 60]. However, SCAF's simple "fitting" is performed repeatedly, adjusting to each round of feedback from the client and reacting dynamically rather than depending on a static model. Such a static model would fail to react to changes in scalability over time, and would require profiling the entire application beforehand.

The above model works well since C_j can be adjusted to approximate speedup curves of real applications using only a single measurement representing recent efficiency. A dynamic system which fits using multiple distinct speedup points might overfit to the application's current behavior, and will react less quickly to changing behavior.

Next, we discuss exactly how the daemon arrives at such allocations. Using the speedup model in equation 4.2, the SCAF daemon is faced with the optimization problem

$$\max_p \left\{ \sum_{i=1}^{k} S_i(p_i) \middle| p > 0 \bigwedge \sum_{i=1}^{k} p_i = N \right\} \tag{4.3}$$

or, equivalently using equation 4.2,

$$\max_p \left\{ \sum_{i=1}^{k} 1 + C_i \log p_i \middle| p > 0 \bigwedge \sum_{i=1}^{k} p_i = N \right\} \tag{4.4}$$

Let Q_i be defined as $Q_i = \frac{C_i}{\sum_j C_j}$. Since $\sum_j C_j$ (a sum of constant quantities) and 1 are constant quantities, we can equivalently express our optimization problem

as

$$\max_{p} \left\{ \sum_{i=1}^{k} Q_i \log p_i \middle| p > 0 \bigwedge \sum_{i=1}^{k} p_i = N \right\} \tag{4.5}$$

Next, we define P_i to be $\frac{p_i}{N}$, such that $\sum_i P_i = 1$. Since $\frac{1}{N}$ is constant, we can express our optimization problem as

$$\max_{P} \left\{ \sum_{i=1}^{k} Q_i \log P_i \middle| P > 0 \bigwedge \sum_{i=1}^{k} P_i = 1 \right\} \tag{4.6}$$

We can obtain an equivalent minimization problem by taking the negative of the objective function. Then, we add the constant quantity $\sum_i Q_i \log Q_i$, resulting in

$$\min_{P} \left\{ \sum_{i=1}^{k} Q_i \log Q_i - \sum_{i=1}^{k} Q_i \log P_i \middle| P > 0 \bigwedge \sum_{i=1}^{k} P_i = 1 \right\} \tag{4.7}$$

or, after combining sum terms,

$$\min_{P} \left\{ \sum_{i=1}^{k} Q_i \log \frac{Q_i}{P_i} \middle| P > 0 \bigwedge \sum_{i=1}^{k} P_i = 1 \right\} \tag{4.8}$$

If we now interpret Q and P as discrete probability distributions, we see that equation 4.8 describes the relative entropy of of Q with respect to P, or the Kullback-Leibler divergence of P from Q. This relative entropy is known to be always non-negative, and equals zero only if $Q = P$. Therefore, the single optimal solution is at $P = Q$. In other words, $p_i = NQ_i$, or

$$p_i = \frac{NC_i}{\sum_j C_j} \tag{4.9}$$

As per equation 4.9, the SCAF daemon sets the allocations p by computing the vector C, then assigning p_i to be the fraction $C_i / \sum C$ of N. Of course, the real allocation needs to be an integer, and starvation must avoided by ensuring that no

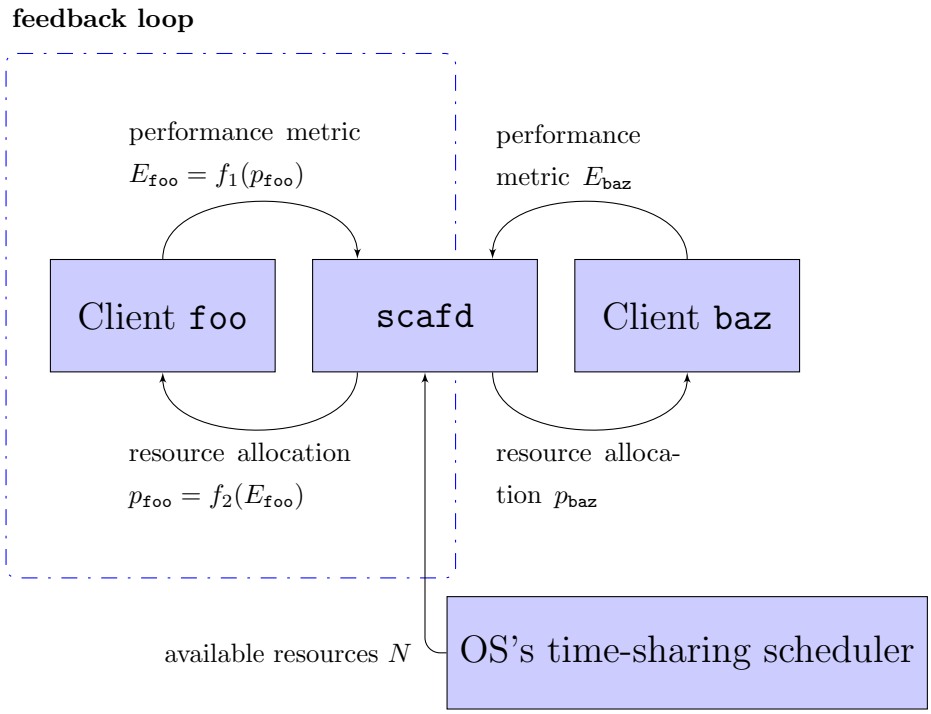


Figure 4.2: Runtime feedback loop in SCAF's partitioning scheme

process receives an allocation less than 1. Section 5.1 describes how p is used to create partitions satisfying these requirements.

It can be shown that had our assumed speedup model been linear instead of logarithmic, then the optimization problem becomes immediately uninteresting: the optimal solution is always to allocate the entire machine to the single process with the greatest speedup function slope, starving other processes. This would be an undesirable allocation.

Figure 4.2 illustrates the feedback loop design in SCAF. Consider a machine with 16 hardware contexts. Say process `foo` is observed to scale fairly well, achieving an efficiency of 2/3 on 12 threads, while `baz` is observed scaling poorly, achieving an efficiency of only 3/8 on 4 threads. The SCAF daemon, applying the speedup

model and solving the optimization problem, will arrive at $C_{\texttt{foo}} = \frac{8-1}{\log 12}, C_{\texttt{baz}} = \frac{3/2-1}{\log 4}$ and compute new allocations $p_{\texttt{foo}} = \lfloor 14.18 \rfloor = 14, p_{\texttt{baz}} = \lfloor 1.82 \rfloor + 1 = 2$. (Note the actual strategy for converting precise allocations to integers is simplified here: see Section 5.5 for more details.) If the resulting feedback indicates a good match with the predicted models, then the same model and solution will be maintained, and allocations will remain the same. If one or more feedback items indicate a bad fit, either due to a change in program behavior or poor modeling, then a new model will be built using the new feedback information. For example, if `foo` scales better than the $1 + \frac{8-1}{\log 12} \log 14 = 8.43$X speedup anticipated by the previous model, then a new model will be created accordingly, and `foo`'s allocation will increase further.

For comparison with Max-IPC (Figure 4.1), equipartitioning (Figure 2.4), gang scheduling (Figure 2.3), an unmodified fine-grained time sharing system (Figure 2.2), and exclusive scheduling (Figure 2.1), please refer to Figure 4.3, which depicts our CG+LU scenario running with space sharing allocated via SCAF's max-speedup policy. Finally, Table 4.3 summarizes all of the scheduling techniques discussed in this paper thus far, including the three space-sharing allocation policies discussed in this chapter.

In Table 4.3, the rightmost column shows the average efficiency of the entire system during the multiprogrammed period of runtime. Because we are computing efficiency over all 36 of the TileGX's cores, one can equivalently think of average efficiency as the average speedup obtained for each core on the system. Please note that the speedups reported in this table are all *effective* speedups based on serial runtime divided by parallel runtime: they are not based on the internal speedup or

efficiency estimates that SCAF's runtime feedback mechanism uses.

Exclusive scheduling, which is not a form of multiprogramming, will have the efficiency equal to that of each program running individually on the entire system. The whole-system efficiency of CG is relatively high, at 0.80, while LU only achieves an efficiency of 0.62. Due to hardware contention, the unmodified system (utilizing fine-grained time sharing of threads) suffers from low speedups and low system efficiency. Gang scheduling effectively averages the efficiencies of the two processes (compared to exclusive runs) because it is simply toggling between the two in time. Equipartitioning notably *improves* on this average due to the fact that each process is now running on fewer hardware contexts via space sharing, which in general results in higher efficiency. Next we see the experimental "Max-IPC" policy applied to space sharing. The only difference between equipartitioning and max-IPC is that a larger fraction of the 36 cores is given to LU in max-IPC. The result is an increase of 8X speedup for LU, but a larger decrease of 13X speedup for CG, leading to a lower sum of speedups and lower system efficiency. Finally, we see the SCAF system, which allocates for space sharing according to an optimization goal of maximizing sum of speedups. With SCAF, rather than giving LU more threads due to its higher IPC, we allocate more threads to CG because of its higher observed efficiency. The result (when compared to equipartitioning) is a decrease of only 4X speedup for LU, and a larger 6.2X increase for CG, leading to an overall improvement in sum of speedups and system efficiency.

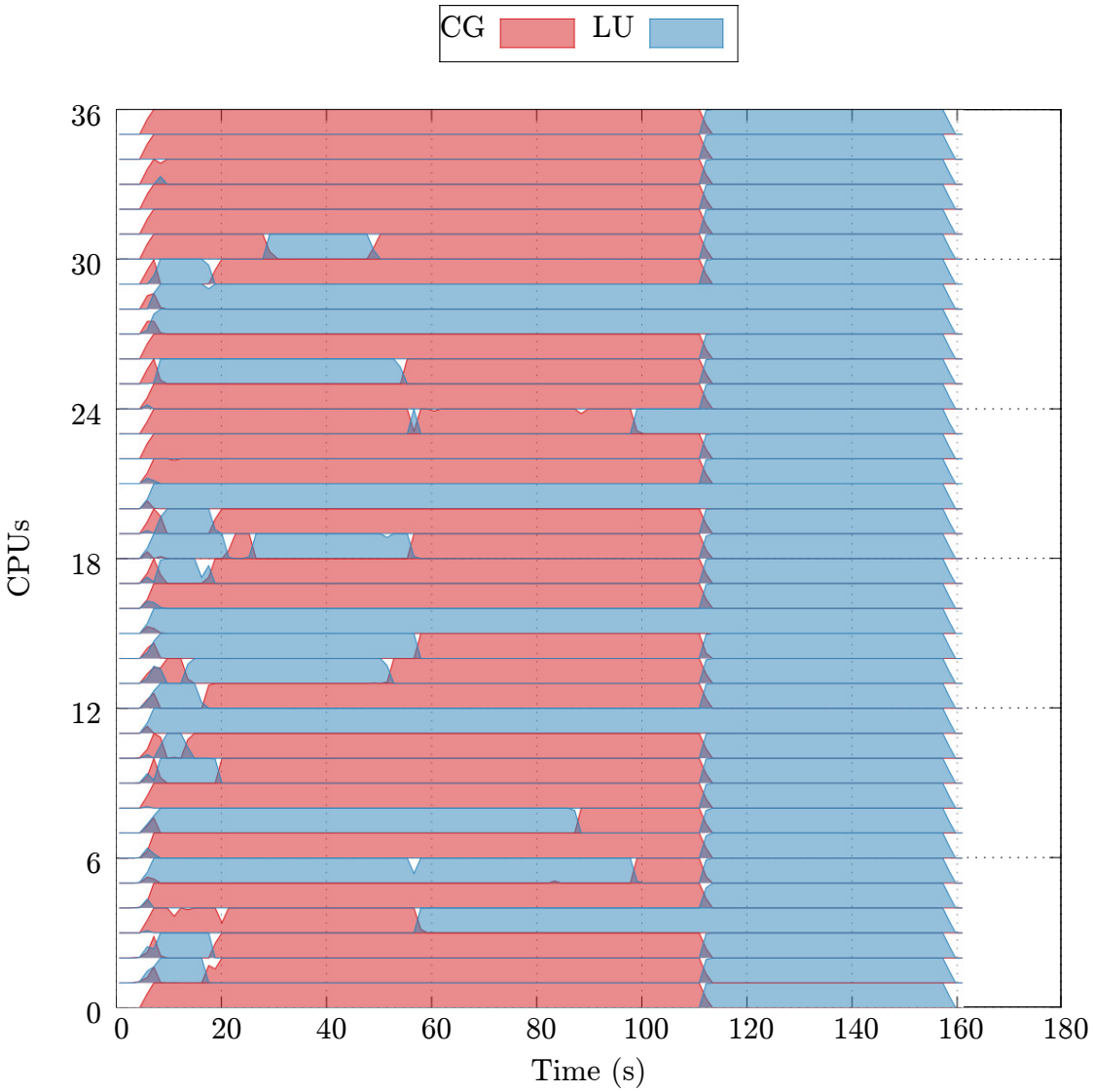


Figure 4.3: Space sharing allocated for maximum system speedup via SCAF on TileGX/Linux

Technique	Process Allocation Policy	Sharing Mechanism	CG Speedup	LU Speedup	$\sum$ Speedup	36-core Efficiency (speedup/core)
Exclusive scheduling	None	None	28.9X	22.4X	(N/A)	0.80, then 0.62
"Unmodified"	None[a]	Fine-grained, time sharing	6.4X	4.1X	12.5X	0.35
Gang scheduling	Equal[b]	Coarse-grained, time sharing	15.6X	10.8X	26.4X	0.73
Equipartitioning	Equal	Space sharing	17.0X	12.0X	29.0X	0.81
"Max-IPC"	Max-IPC	Space sharing	4.0X	20.0X	24.0X	0.67
SCAF	Max-speedup	Space sharing	23.2X	8.0X	31.2X	0.87

Table 4.3: Summary of all multiprogramming techniques in CG-LU scenario, including 3 space sharing policies

[a]OS thread schedulers generally have many sophisticated scheduling policies available, but intentionally tend to deal with *threads* rather than processes.

[b]More sophisticated gang-scheduling policies are possible by varying the duty cycle of each process. However, we refer to only the most generic case here: other variations suffer from the same disadvantages for our purposes.

Chapter 5

Implementation

SCAF easily integrates into existing systems without requiring modification or recompilation of programs by providing SCAF-aware versions of parallel runtime libraries. Specifically, our implementation supports OpenMP programs compiled with the GNU Compiler Collection as clients.[1] Just before execution begins, such programs load a shared object which contains the OpenMP runtime, `libgomp` [61]. A user or administrator can specify that the SCAF-aware version of the runtime should be used, making any subsequently launched processes SCAF-aware. SCAF-aware and traditional processes may coexist without issue.

The SCAF-aware implementation of `libgomp` is a one-time modification of the original, involving only a few lines of changed code and about two days of graduate student time. (Mostly spent understanding `libgomp`.) These minor changes call into a `libscaf` client library which is designed to easily support development of additional runtime ports with similar ease. Intel's `libiomp5` has also been ported. Currently, the `libscaf` library itself consists of less than 2000 lines of C.

Although SCAF supports all malleable OpenMP programs, it is important to note that not all OpenMP programs are malleable. Specifically, the OpenMP

[1] We have also ported the Intel OpenMP Runtime because the Xeon Phi requires the use of Intel's compiler for good performance. The changes required are the same as those required for `libgomp`.

standard permits programs to request or explicitly set the number of threads in use by the program [62]. Programs that make use of this functionality are assumed by SCAF to be non-malleable, since they may depend on this number. An example of a C OpenMP program with this non-malleable behavior is shown in Listing 5.1:

Listing 5.1: OpenMP code which depends on the number of threads initially used

```
/* Here the code notes the number of initial threads. */
unsigned num_threads = omp_get_max_threads();
float **buckets = malloc(sizeof(float*) * num_threads);
for(i=0; i<num_threads; i++){
    buckets[i] = malloc(sizeof(float) * BUCKETSIZE);
}

for(i=0; i<1024; i++){
#pragma omp parallel for
    for(j=0; j<1024; j++){
        /* Here, the parallel loop is expecting    *
         * that the number of parallel threads has *
         * not changed.                            */
        process_bucket(buckets[omp_get_thread_num()])
    }
    aggregate_buckets(buckets, num_threads);
}
```

In Listing 5.1, "buckets" are set up as temporary processing buffers, one per thread. Before any parallel loops are run, the total number of threads is recorded via `omp_get_max_threads()`. A number of temporary buffers are allocated according to this number, and then during the execution of parallel loops, threads are expected to populate these buffers. If the number of threads executing the parallel loop (i.e., executing the iterations of `for(j)`) deviates from `num_threads`, then this may result in `buckets` being populated incorrectly. Therefore, while this program can *begin* execution on any number of threads, it must not change the number of threads during execution, and it is implicitly not malleable.

Since SCAF implements the client's OpenMP interface, it can detect when a non-malleable program requests the number of threads, and simply consider that application's thread count to be fixed after that point. As a result, SCAF is safe to use as a drop-in replacement for GNU OpenMP on a system even if the system runs a mixture of malleable and non-malleable OpenMP applications.

Finally, we would like to note that implicitly non-malleable code such as in Listing 5.1 is technically not portable to all OpenMP runtimes, since the OpenMP specification explicitly allows OpenMP implementations to change the number of threads unless `omp_set_dynamic(0)` is used by the programmer to indicate that it is not allowed. As a result, an OpenMP implementation using SCAF does not violate any specifications even without SCAF fixing the number of threads to the value returned by `omp_get_max_threads()`.

5.1 The SCAF Daemon

Listing 5.2: Running the `scafd` SCAF daemon, printing status every 1 second

```
$ scafd -t 1
PID       NAME      THREADS  NLWP
all       -         16       -
PID       NAME      THREADS  NLWP
all       -         16       -
1234      foo       4        4
5678      bar       12       12
```

The system-wide SCAF daemon, `scafd`, communicates with the SCAF clients using a portable software bus, namely ZeroMQ [63]. For the sake of portability, the SCAF daemon is implemented entirely in userspace. The `scafd` implementation is

itself multithreaded for the sake of concurrency, with various threads performing distinct tasks. While the SCAF daemon could run on a separate host, it incurs a small enough load that this is not necessary. The SCAF daemon has five jobs: 1) monitor load on hardware contexts due to uncontrollable processes, 2) maintain and monitor the list of malleable clients, 3) compute the hardware context partitioning using runtime feedback from the clients, 4) service requests from SCAF clients for their current hardware context allocation, and 5) print or log its status to standard output. In this section, we describe the implementation of these five functions.

1. Uncontrollable (non-malleable) process load is monitored through the operating system's facilities in a "lookout" thread. For example, on FreeBSD and Linux, the lookout thread monitors the number of kernel timer interrupt intervals (*i.e.*, "ticks" or "jiffies") which have been used by processes which it does not know to be SCAF-compliant processes and uses this to compute the number of hardware contexts which are effectively occupied. This is the same general, inexpensive method used by programs like `top`. The lookout thread queries the kernel only once per second by default. The lookout thread is also responsible for noticing any client processes which have not reported an efficiency estimate recently, for purposes of detecting long-running sections as described in Section 5.4.

2. The daemon is notified of new clients by their first message over the software bus. A "reaper" thread runs periodically to poll and check if any client processes have exited unexpectedly, so that their allocations may be released. However,

`libscaf` normally uses the POSIX `atexit(3)` mechanism, which allows us to add a cleanup function to the exit of any process without recompilation. Normally, this allows a process to actively notify the SCAF daemon when it exits, and so the reaper is only a backup system. The list of clients and their state are maintained in a single efficient "UTHASH" hash table structure, written by Hanson [64]. The client hash table is then shared among all threads in the `scafd` process, with POSIX threads-based synchronization where exclusive access is appropriate. Note that there is no need for SCAF *clients* to have access to this table: it is only accessible within `scafd`.

3. The partitioning of hardware contexts to processes is performed only periodically at a tunable rate, completely asynchronously from clients' requests, by the "referee" thread. By default, the referee computes partitions as described in section 4.3. Algorithm 2 describes `scafd`'s computation of new allocations. Note that the conversion of exact allocations to integers is a surprisingly non-trivial process which required the implementation of a small library, `intpart`, which is distributed with SCAF. `intpart` and its interfaces are described in Section 5.5.

   An alternative referee which implements equipartitioning is included in the `scafd` implementation, since there is no other publicly-available solution for equipartitioning that we are aware of. The equipartitioning referee can be requested of `scafd` either by specifying the `-e` option upon launch, or by sending `SIGUSR1` to the `scafd` process after launch. (This can be done while clients

are running.) Algorithm 3 describes `scafd`'s computation of equipartitioned allocations.

4. Requests from SCAF clients, received over the software bus, arrive in the form of a message containing the client's most recent efficiency metric. These messages are received by the parent `scafd` thread. This information is stored immediately but not acted upon immediately, since it arrives at a high rate. In order to respond to the requests at the same rate, the daemon periodically evaluates the stored set of client efficiencies and computes a new set of hardware context allocations. This scheme allows the daemon to respond immediately to any requests by returning the latest computed allocation, which may not have incorporated the very latest reported client measurements yet. Other than the initial message a client sends to announce itself to `scafd`, this is the only kind of communication necessary between the clients and the daemon. The rate at which the daemon computes new allocations is tunable, and defaults to 4 Hz. The rate at which clients check in is variable, depending on the duration and frequency of parallel sections in their progams, but generally much higher than 4 Hz in our test suite.

5. Finally, a "scoreboard" thread optionally runs at a chosen interval and prints current allocations and metrics deemed useful for analyzing `scafd` behavior.

Algorithm 2: `scafd`'s allocation computation algorithm, targeting max-speedup

while *sleep* $\frac{1}{4}$ *seconds* **do**

$sumC \longleftarrow 0$

$rwLock(clients)$ `// exclusive access to client structures`

$numClients \longleftarrow hashCount(clients)$

for $current \in clients$ **do**

$C_{current} \longleftarrow \frac{E_{current} \times p_{current} - 1}{\log(p_{current})}$ `// Compute coefficients`

$sumC \longleftarrow sumC + C_{current}$

for $current \in clients$ **do**

$exactAlloc_{current} \longleftarrow C_{current}/sumC$ `// Normalize` C

$availThreads \longleftarrow N - \lceil uncontrollableUsage \rceil$

$\overline{\hat{p}} \longleftarrow intpartFromFloatpart(availThreads, numClients, exactAlloc)$

$unlock(clients)$ `// release lock and sleep`

Algorithm 3: `scafd`'s alternate allocation computation algorithm, targeting equipartitioning. No runtime feedback is used.

while *sleep* $\frac{1}{4}$ *seconds* **do**

$rwLock(clients)$ `// exclusive access to client structures`

$numClients \longleftarrow hashCount(clients)$

$availThreads \longleftarrow N - \lceil uncontrollableUsage \rceil$

$\overline{\hat{p}} \longleftarrow intpartEquipartition(availThreads, numClients)$

$unlock(clients)$ `// release lock and sleep`

5.2 The `libgomp` SCAF Client Runtime

The meat of the SCAF implementation lies in the `libscaf` client library, but for the sake of clarity it is described here in the context of the `libgomp` client runtime which uses it. The clients perform three interesting functions: 1) recording baseline serial IPC using lightweight serial experiments, 2) recording parallel IPC, and 3) computing parallel efficiency relative to the experiment results as the program runs.

5.2.1 Lightweight Serial Experiments

Figure 5.1 illustrates the lightweight serial experiment technique. In SCAF, a lightweight serial experiment allows the client to estimate the serial performance of a parallel section of code. This allows the client to then compute its recent efficiency, and provide a meaningful metric to the SCAF daemon. By default, the client will perform an experiment *only the first time it executes each parallel section, thus*

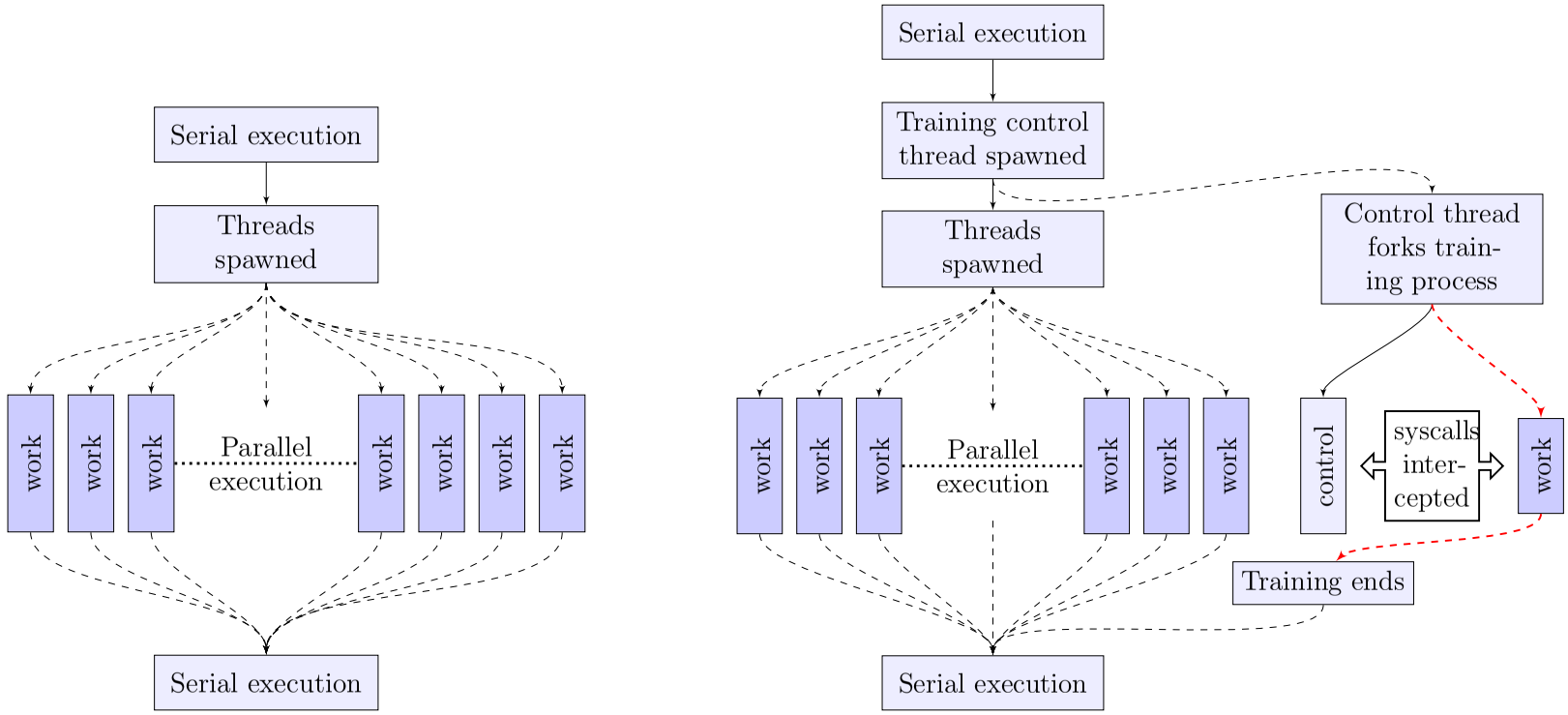
reducing its overhead, although the user is able to tune the client so that it re-runs the experiment periodically. Experiments proceed as follows: given an allocation of N hardware contexts to run parallel section A on for the first time, the `libscaf` will recognize that it has no serial experiment result for A and is due for an experiment run. Provided a function pointer to the parallel section, `libscaf` creates a new *process* which will run the parallel section A serially on a single hardware context concurrently with the original parallel process. Although the experimental process is a separate proper process, it must share the allocation of N hardware contexts in order to avoid oversubscription. Oversubscription would likely hurt performance, but more importantly, it would almost certainly adversely affect the experiment's measurements. To accomplish this, `libscaf` simply reduces the number of hardware contexts on which the non-experimental process runs on to $N - 1$. The end result is an experimental process running on 1 thread for the sake of measuring its achieved IPC, while the original program still makes progress as usual with $N - 1$ threads. Note that the serial execution of the section is not timed, since it may be interrupted early. Instead, its IPC is recorded, since this will still be meaningful.

The act of creating a new process with the state and instructions required to proceed as an experimental process may sound technically daunting, unreliable, or possibly highly specific to `libgomp`. We assure the reader that this couldn't be further from the truth. All that is required to implement this is the ability to set the number of threads to 1, and the `fork(2)` system call. Any parallel runtime which supports malleable processes necessarily supports setting the number of threads to 1. `fork(2)` is actually the mechanism by which most processes on a UNIX-like system

are created, which virtually ensures its availability. Finally, we note that `fork(2)` implements nearly all of the steps required for running an experimental copy of a process since the child process begins life as a single threaded with exactly identical memory and process state to the parent. Therefore, at a high level, all that is needed for experiment creation is for the parent process to set the number of threads to 1, ensure that the experiment will stop when interrupted or completed, and then call `fork(2)` to begin the experiment.

Experiment duration Assuming some speedup is being achieved, the serial experiment process would take longer to complete than the parallel process doing the same work. We cannot afford to wait that long. Thus, we end the experimental process as soon as the parallel process finishes the section. The experiment is interrupted using POSIX signals, and then the experimental result is communicated back to the parent process using ZeroMQ. The achieved IPC of the serialized section is recorded by the parent in order to compare it to parallel IPC measurements.

Maintaining correctness Since there will be two instances of the original section in execution, care must be taken to avoid changing the machine's state as perceived by its users. The forked experimental process begins as a clone of the original parallel process just before the section of interest. The new process's memory is a copy of the original process's, so there is no fear of incorrectly affecting the original process through memory operations. The only other means a process has to affect system state is through the kernel, by way of system calls. For example, `printf(3)`, `fprintf(3)`, and other high-level functions must eventually use the kernel's `write(2)` system call to write to a file or terminal. Similarly, `connect(2)` must



(a) Parallel section without a lightweight serial experiment (b) Parallel section with a lightweight serial experiment

Figure 5.1: Illustration of lightweight serial experiments

be used in order for any communication via sockets or the network. Other system calls, such as `gettimeofday(2)` or `getuid(2)` do not affect the machine's state, and can be allowed within the experiment. Fortunately, `ptrace(2)` on platforms such as FreeBSD and Linux provides a mechanism for intercepting and denying system calls selectively. On Solaris, the `proc(4)` filesystem can be used to the same effect. Therefore, the experimental process runs until an unsafe system call is requested. For example, a read from a file descriptor is allowed. A series of writes may be allowed, but only if the write is redirected to `/dev/null`. (Nowhere.) A series of writes followed by a read is not allowed, as the read may be dependent on the previous writes, which did not actually occur. Fortunately, we have found that the sophistication of the runtime's system call filtering is rather unimportant: parallel sections tend to contain few system calls, and terminating experiments due to unsafe system calls is the exception rather than the norm. For example, none of the NAS NPB benchmarks contain such unsafe system calls in their parallel sections.

Performance of `fork()` On modern UNIX or UNIX-like OSs, `fork` only copies the page table entries, which point to copy-on-write pages. This avoids the penalty associated with allocating and initializing a full copy of the parent's memory space. As a result, `fork` is still more expensive than thread initialization, but is not prohibitively expensive when used infrequently for serial experiments. See section 6.2.3 for more in-depth discussion of virtual memory management's importance to SCAF.

5.2.2 Computing Efficiency

The SCAF runtime calculates an effective efficiency in order to report it back to the SCAF daemon before each parallel section. The client receives an allocation of N threads, which it uses in order to compute the next parallel section. This allocation is considered fixed across any serial execution that occurs between parallel sections. In the OpenMP port, the client constantly collects five items in order to compute its reported efficiency:

1. T_{parallel} : wall time spent inside the last parallel section
2. P_{parallel} : the per-thread IPC recorded in the last parallel section
3. T_{serial} : wall time spent after the last parallel section executing serial code
4. S : an identifier for the last parallel section, generally its location in the binary
5. N : the thread allocation used since the start of the last parallel section

Here it is important to note that T_{serial} **and** T_{parallel} *refer to time spent in different work; in particular, non-parallelized OpenMP code and explicitly parallelized OpenMP code, respectively*, and not to time spent performing the same work. That is, T_{serial} is not related to any lightweight serial experiment measurements.

The client then can compute the following efficiencies, given that it has the serial IPC $P(S)$ of S from a completed lightweight experiment:

$$E_{\text{parallel}} \leftarrow P_{\text{parallel}}/P(S) = \frac{P_{\text{parallel}}N/P(S)}{N} \approx \frac{\text{speedup}}{N}$$

$$E_{\text{serial}} \leftarrow 1/N \approx \frac{\text{speedup}}{N}$$

Since processes report efficiencies only at the beginning of each parallel section, thus $T_{\text{parallel}} + T_{\text{serial}}$ is the time since the last efficiency report to the SCAF daemon. Efficiency since the last report is then estimated as

$$E_{\text{recent}} \leftarrow \frac{E_{\text{serial}} \cdot T_{\text{serial}} + E_{\text{parallel}} \cdot T_{\text{parallel}}}{T_{\text{serial}} + T_{\text{parallel}}}$$

Finally, before being reported to the SCAF daemon, this efficiency value is passed through a simulated RC low-pass filter, with adjustable time constant RC:

$$E \leftarrow \alpha \cdot E_{\text{recent}} + (1 - \alpha) \cdot E, \text{ with}$$

$$\alpha \leftarrow (T_{\text{serial}} + T_{\text{parallel}})/(RC + T_{\text{serial}} + T_{\text{parallel}}).$$

This is a simple causal filter which requires only one previous value to be held in memory. This keeps the efficiency rating somewhat smooth, but at the same time does not punish a process for performing poorly in the distant past. The hope is that the recent behavior of the program will be a good predictor for its behavior in the near future.

5.3 Supporting Non-malleable Clients

While the preferred mechanism for controlling parallelism with SCAF is to change the number of busy threads in processes, SCAF can also support non-malleable OpenMP programs. Non-malleable programs are by definition those whose number of threads cannot be changed at run-time by the scheduler. Hence SCAF cannot change the number of threads to avoid the overheads of time sharing for that process. For non-malleable programs we present an alternative approach which

relies on keeping the number of threads unchanged, but changing their affinity to specify how many hardware threads the process can run on. We outline this method below, including how it impacts the various aspects of SCAF, including controlling parallelism, lightweight serial experiments, recording parallel IPC, and computing parallel efficiency.

Instead of changing the number of threads that exist in a non-malleable process in order to control parallelism, `scafd` changes the number of threads from that process which are allowed to run simultaneously. For example, if a non-malleable process "`foo`" is running on a machine with 16 hardware contexts, it will consist of 16 threads. Another process, "`bar`" begins execution. If `foo` were malleable, we could eliminate eight of its threads, leaving 8 hardware contexts unused for `bar`. Instead, since `foo` is not malleable, we use OS facilities to administratively restrict it to using only the first eight hardware contexts of the machine. (We specify that it should have an "affinity" for those contexts.) Specifically, both Solaris and Linux allow a process and all of its threads to be limited to a given set of hardware contexts; it is not necessary to specify which hardware context each particular thread should use. This allows `bar` to use the other 8 contexts with no competition, effecting space sharing on the machine. We note that this method of controlling parallelism should not be used unless necessary, since it will incur oversubscription within the eight hardware contexts that `foo` is running on. Another subtle disadvantage to this mechanism is that it requires SCAF to specify the particular set of hardware contexts to restrict the process to, and we want to leave this up to the OS's thread scheduler as much as possible. However, this is a simple way of safely effecting space-sharing in the

presence of non-malleable processes, so we use it.

Serial experiments within non-malleable processes are also possible. Consider the case of `foo` and `bar` again, where `foo` is non-malleable. In order to run a serial experiment on `foo`, we shrink the number of allowed contexts used by the main `foo` process by one, leaving one explicitly available for the serial experiment. This prevents the experiment's run from being subject to fine-grained time sharing with the 16 parallel threads making normal progress in `foo`, and results in a more realistic experiment.

Finally, the computation used to estimate parallel IPC and efficiency must be modified slightly for `foo` in order to account for the fact that it has more threads than hardware contexts allocated. Instead of computing $P_{parallel}$ as the per-thread IPC for the last parallel section, $P_{parallel}$ must represent the IPC achieved per *hardware context*. For example, if `foo` achieves 0.1 IPC during a parallel section on each of its 16 threads, $P_{parallel}$ should be 0.2 IPC. This allows `foo`'s reported efficiency to reflect the entire process's performance. We note that if `foo` does suffer a large slowdown due to its contained oversubscription, this should be reflected in its reported efficiencies, allow SCAF to reduce its allocation accordingly.

5.4 Supporting Long-running Parallel Sections

In some cases, an OpenMP program is malleable, but has long-running parallel sections. In other words, the entry points of parallel sections (the only points at which SCAF can change the number of threads in the process) may come only

infrequently, or only once. In the worst case, a program may consists of one single long-running parallel section, giving SCAF no chance to use experimental results or evaluate parallel efficiency or even instruct the parallel runtime to change the number of running threads.

We support these programs by way if a timeout mechanism in `scafd`: if `scafd` notices that a program began a parallel section but has not been heard from again in more than 10 seconds, it will pretend as though the process is non-malleable. This allows SCAF to safely control the parallelism of the program using the same affinity-based mechanism described in section 5.3.

However, one problem remains: how can we obtain performance feedback as the program runs given that control is never returned to the SCAF code in the client runtime? A serial experiment that has been launched will never never end, because the SCAF code that normally runs at the end of parallel sections does not get a chance to run.

We solve the problem of obtaining feedback within a long-running section by asynchronously injecting feedback collection points into the section. When `scafd` notices it hasn't heard from a process in 10 seconds, it will send a POSIX signal to the process, which invokes a pre-configured signal handler in the unresponsive process. The signal handler first ends any running experiments, gathers its result, and uses it to estimate the long-running parallel section's efficiency thus far. It replies to `scafd` after computing the process's estimated efficiency, and then returns control to the interrupted parallel section. Essentially, the signal handler behaves as a SCAF-injected, very short serial section, which runs only in order to report

efficiencies. Because the parallel section must resume running on its original number of threads, control over parallelism must be done using the affinity-based mechanism described in section 5.3. As the long-running section continues, `scafd` may send the signal periodically in order to continue to receive feedback.

5.5 The `intpart` library

In this section we describe the functionality provided by the `intpart` library, whose source code is distributed with SCAF. `intpart` accepts a specific precise partitioning $\overline{p}$ of 1.0 and an integer N, and outputs the single *integer* partitioning $\overline{\hat{p}}$ of N which best approximates the ratios seen in $\overline{p}$. This is an important problem to SCAF because SCAF implements a feedback loop: integer allocations ($\overline{\hat{p}}$) should reflect the optimal floating point solution ($\overline{p}$) as much as possible in order to provide and responsive useful feedback. The original implementation of SCAF used truncation to convert precise floating point allocation ratios to integer core allocations, allocating any leftover cores to allocations which were truncated the most. However, as testing in more scenarios went on and "chunk" sizes were introduced (see Section 6.2), it became apparent that a naive conversion of floating point to integer partitions was affecting the feedback loop. `intpart` was written to solve this general problem.

`intpart` primarily implements the *intpartFromFloatpartChunked* function, which generates the integer partitioning subject to a given *chunkSize*. Both the input (floating point) and output (integer) partitions are of cardinality l. Very high-level pseudo-code for this routine can be found in Algorithm 4.

Algorithm 4: Pseudo-code for the `intpart_from_floatpart_chunked` routine in `intpart`

$remainingChunks \longleftarrow N/chunkSize$

$\overline{\hat{p}} \longleftarrow \overline{p} \times remainingChunks$ `// Initial crude truncation`

Compute vector of errors between $\overline{\hat{p}}$ and $\overline{p}$

while $remainingChunks > 0$ **do**

  if *an empty partition exists* **then**

    $needy \longleftarrow$ partition index of first empty partition

  else

    $needy \longleftarrow$ partition index with greatest error

  $remainingChunks \longleftarrow remainingChunks - 1$

  Compute vector of errors between $\overline{\hat{p}}$ and $\overline{p}$

while *Empty partitions exist in* $\overline{\hat{p}}$ **do**

  Steal chunks from the non-empty partition with least error

  Compute vector of errors between $\overline{\hat{p}}$ and $\overline{p}$

$\overline{\hat{p}} \longleftarrow \overline{\hat{p}} \times chunkSize$

return $\overline{\hat{p}}$

Chapter 6

Adaptation to Various Platforms

While adapting SCAF to several diverse platforms, we have discovered that the key to significant improvements with SCAF is to make the machine appear to scale in a simple, predictable manner. If a class of computer does not naturally scale in this manner, we present methods to adapt SCAF, or the machine's behavior, to make the machine appear as if it does scale in a simple, predictable manner.

Table 6.1 enumerates the platforms used and their key features. We selected these platforms in an effort to explore a wide set of system architectures with different approaches toward memory access, exploiting parallelism, and interconnects.

The Tile-GX is a 6x6 grid of small single-thread cores, using a mesh network for cache coherence and communication with two memory controllers located at the edges of the grid [65]. The Xeon E5-2690 system consists of two sockets, each with its

Machine	Processors	NUMA	Hardware Multithreading	Interconnect	Out-of-order Execution	Cores	Threads
tilegx	Tile-GX 8036	2 nodes	No	Mesh	No	36	36
openlab08	2x Xeon E5-2690	2 nodes	No (disabled)	P2P	Yes	16	16
tempo-mic0	Xeon Phi 5110p	No	Simultaneous	Ring-bus	No	60	240
triad	UltraSparc T2	No	Temporal	Crossbar	No	8	64
bhindi	2x Opteron 6212	4 nodes	Simultaneous	P2P	Yes	8	16

Table 6.1: Summary of platforms used to evaluate SCAF

Characteristic	Hinders SCAF?	Reason	Fix	Impact of Fix	Evidence
Temporal multithreading	No	Needs TLP; usually scales monotonically			triad vs bhindi, tempo-mic0
Simultaneous multithreading	Yes	IPC per thread unpredictable	allocate in "chunks"	more predictable, still low potential for improvement	tempo-mic0, bhindi
NUMA	No	Sufficient interconnect			openlab08 vs tilegx, t2
OoO execution	No	Affects thread IPC uniformly			openlab08 vs tilegx, tempo-mic0
Slow TLB flushes	Yes	Slows copy-on-write and experiments	Linux 3.6+ `tlb.c`	faster `fork`; overhead vastly decreased	tempo-mic0
"Home" cache movement on `fork`	Yes	Threads permanently slower after experiments	Improved DDC heuristics in Linux MM	homes preserved; slowdown eliminated	tilegx

Table 6.2: Summary of platform characteristics and their impact on SCAF

own set of 8 single-threaded cores, memory controller and main memory [66]. Within a socket, a ring bus is used for coherence communication, while a point-to-point link is used for coherence and remote memory access between sockets. The Xeon Phi consists of 60 lightweight cores connected to one another and memory controllers via a ring-bus network [67]. The UltraSparc T2 system consists of 8 lightweight cores connected to on-chip memory controllers with a crossbar network [68]. Finally the Opteron 6212 system consists of two sockets, each with two memory controllers and four "modules", where each module consists of a sophisticated out-of-order core with two integer pipelines [69]. Please see Figures A.1-A.5 in the appendix for more detailed topology information.

Table 6.2 summarizes our findings across all platforms, including both hardware features and notable operating system issues. The ideas summarized in Table 6.2 are detailed next in sections 6.1 and 6.2.

6.1 Hardware Characteristics

Although SCAF assumes a simple speedup model, there are of course many features of real platforms which can result in more complex speedup behavior. In this section, we enumerate and discuss the important features we encountered in our evaluation of SCAF on five distinct platforms, including representative speedup curves to illustrate machine behavior. Detailed results on these platforms are presented in chapter 7.

6.1.1 Hardware Multithreading

The largest complication for speedup behavior we have discovered is hardware multithreading. In our test systems, we found that specifically *simultaneous* multithreading is unpredictable, while forms of *temporal* multithreading are less problematic for SCAF. Three of our platforms employ some form of hardware multithreading, each distinct from the others. All three systems are described in this section, which is itself summarized in Table 6.2.

Bhindi Our Opteron 6212 test system, "bhindi," implements a variant on simultaneous hardware multithreading which AMD calls "Clustered Multithreading," or "CMT." In this scheme, each thread has its own integer pipeline and L1 data caches, but the floating point unit, L1 instruction cache, instruction fetch unit, and L2 cache are shared by two threads. For organizational purposes within this paper, we classify CMT as simultaneous hardware multithreading because CMT allows instructions from different threads to be issued in the same cycle.

However, we found that for many of our benchmarks, per-thread performance drops when using both threads in the integer cluster, resulting in the complex speedup behavior seen in Figure 6.1. The Opteron 6212's cores are relatively sophisticated out-of-order cores with deep pipelines and speculative execution, intended to achieve good single-threaded performance. We believe that when both hardware threads are in use, per-thread performance can decrease due to frequent pipeline restarts after invalidating probes hit on speculative loads. For example, if a process is running on 8 threads, its threads may be placed one-per-core by Linux. If we increase the

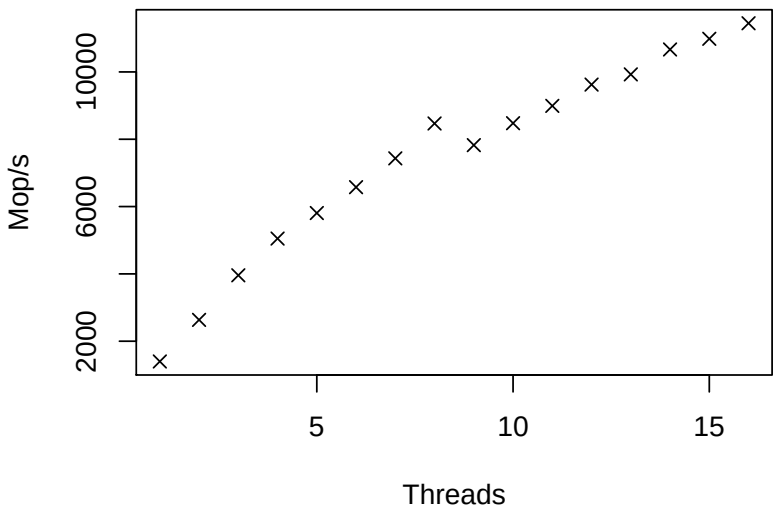


Figure 6.1: FT benchmark scaling on "bhindi" (2x Opteron 6212)

allocation from 8 threads to 9 threads, then Linux must schedule the additional thread to one of the 8 cores, using both of its integer pipelines. Now, 2 of 9 threads are progressing significantly slowly, creating a load imbalance. In some cases the resulting program speedup can actually decrease significantly. For example, FT on 8 threads spread across all cores achieves about 6.7X speedup, but this drops to 5.5X speedup when using 9 threads. At 16 threads, all pairs of integer pipelines are scheduled and FT sees about 8.1X speedup. Comparing the speedups on 8 vs 16 threads, we observe only a small increase in performance by doubling threads. We observe this behavior on several benchmarks, even those that scale well on other computers, showing that the extensive hardware sharing in CMT takes its toll on performance.

The effect of this behavior with SCAF is that benchmarks tend to have a natural decrease in parallel efficiency after the 8-thread mark. This pulls the optimal allocation for 2-way multiprogramming scenarios toward equipartitioning. Addition-

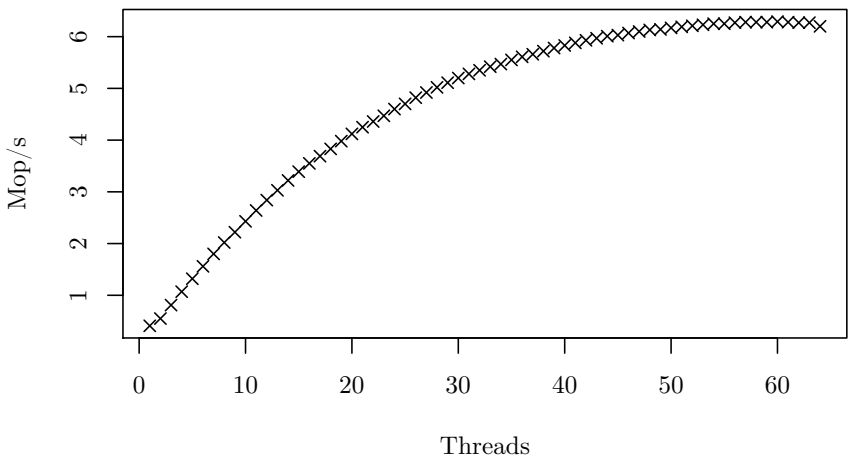


Figure 6.2: UA benchmark scaling on "triad" (UltraSparc T2)

ally, the potential load imbalance means that odd-numbered thread allocations in 2-way multiprogramming are generally undesirable: even if the OS is able to schedule one process across 7 cores, it cannot schedule the other process's 9 threads across cores without scheduling two threads to one of them.

Triad We found that our UltraSparc T2-based system, "triad," generally exhibits simple speedup curves as seen in Figure 6.2. The UltraSparc T2 implements a highly-threaded design with 8 cores, 2 integer pipelines per core, and 4-way temporal multithreading on each integer pipeline. The T2 is not designed to have high single-threaded performance. Thread-level parallelism should be used in order to hide latency to main memory, allowing the use of small, simple, in-order cores. When a thread encounters a long-latency event such as a cache miss, instructions from other threads are issued. We found that running additional threads on a core generally allows the processor to remain busy, and resulted in improved throughput. This works well in the context of SCAF, since allocating additional threads results in increased speedup.

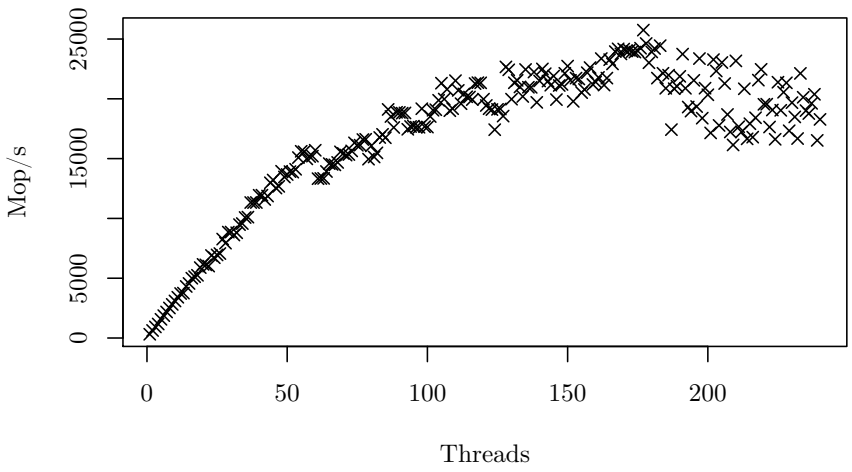


Figure 6.3: MG benchmark scaling on "tempo-mic0" (Xeon Phi 5110p)

Tempo-mic0 Our Xeon Phi test system, "tempo-mic0," often sees a decrease in efficiency beyond 120 out of 240 threads, as seen in Figure 6.3. Most often, this results in optimal allocations (for two-way multiprogramming) near equipartitioning. The Xeon Phi 5110p implements a form of simultaneous multithreading with four hardware threads for each of the 60 cores where only two threads can have instructions issued simultaneously. Due to its two-cycle pipelined instruction decoder, it cannot issue instructions from the same thread in consecutive cycles. However, if there are multiple threads scheduled to the core it will issue 2 instructions per cycle. As a result, while scheduling 1 thread per core will generally scale well, maximum instruction throughput requires at least 1 threads per core. If 3 or 4 threads are scheduled per core, speedup behavior becomes complex, with larger allocations sometimes resulting in very small or no performance gains.

6.1.2 Non-Uniform Memory Access

We found that NUMA did not introduce any unusual speedup behavior, although it is worth pointing out that the machines we consider next in this section are only 2 and 4-node systems. NUMA systems are comprised of NUMA nodes, which each have their own local memory. Some form of inter-processor communication is used for access to non-local memory and cache coherence. It is easy to imagine that such a system might introduce non-monotonic speedup behavior. For example, increasing a process's processing allocation may add non-local processors to those accessing the working set of memory, resulting in a speedup curve that might be lower. Fortunately, we did not observe any major difficulties due to NUMA architectures, thanks to a combination of fast interconnects and NUMA aware thread scheduling.

SCAF specifically avoids dictating placement of threads as much as possible, either to processing units or memory controllers. The affinity-based parallelism control described in section 5.3 does restrict a process to a specific portion of the machine, but the placement of individual threads within that space on the machine is left entirely to the operating system. This allows us to benefit from Linux kernel's numerous techniques and policies for dynamically placing processes near local memory. These techniques are sophisticated enough to handle multiprogrammed scenarios. Because SCAF is only concerned with deciding on the number of threads that processes should be scheduled, the two problems are somewhat orthogonal. While SCAF decides upon the number of threads that the processes should use, the Linux kernel attempts to place the threads close to the memory they are using, potentially

migrating memory allocations between NUMA nodes.

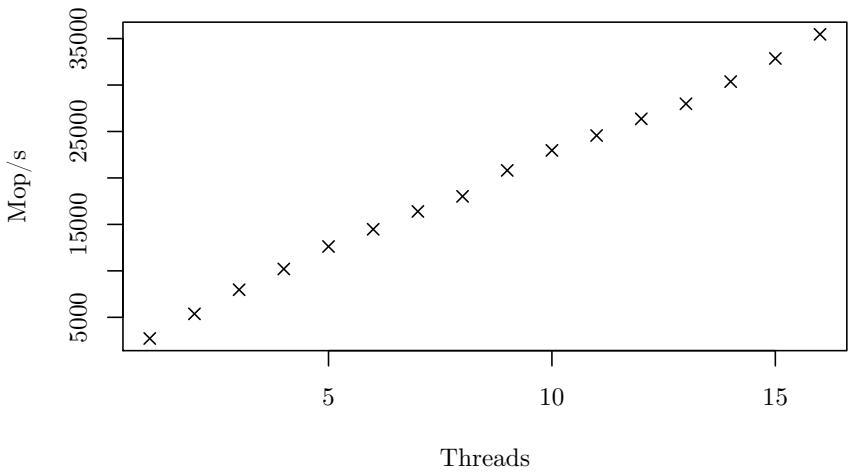


Figure 6.4: BT benchmark scaling on "openlab08" (2x Xeon E5-2690, HyperThreading disabled)

Figure 6.4 shows an example of excellent 2-node NUMA scalability on "openlab08," which has 2 NUMA nodes spread across two sockets. In this speedup curve, the OS is scheduling threads across NUMA nodes as much as possible. The interconnect between NUMA nodes is fast enough that speedup increases linearly with the number of threads. The result is a machine which is very well-behaved and predictable as SCAF modifies allocations. Table 6.2 summarizes our experience with NUMA architectures' impact on SCAF.

6.1.3 Out-of-order Execution

We did not observe any complications due to out-of-order execution with our test machines. Although out-of-order execution can have varying benefits depending on the code being run, the benefits tended to be generally uniform across the hardware contexts in the machine. For example, the Tile-GX and the Xeon E5 systems both have two memory controllers and no multithreading. One major way in which they

differ is that the Tile-GX has simpler cores with in-order instruction execution, while the Xeon E5 employs extensive out-of-order techniques. The Xeon E5's per-thread performance is of course dramatically higher, but both exhibit generally simple scalability. As a result, we believe that the presence or absence of out-of-order execution alone does not adversely affect the effectiveness of SCAF.

6.2 Optimizations

6.2.1 Avoiding Bad Allocations

As described in section 6.1.1, the most problematic hardware feature we encountered was simultaneous multithreading. SMT causes unpredictable speedup behavior because of the fact that individual thread performance depends greatly on a combination of the program being executed and the operating system's decisions regarding where to place threads. This problem can be alleviated somewhat by avoiding certain "bad" allocations, based on knowledge of the topology of the target system.

For example, we found on our Opteron-based test system that allocations of odd numbers of threads are essentially never beneficial because of the effects of 2-way SMT as described in section 6.1.1. More importantly, odd-sized allocations are *unpredictable*. Due to our knowledge of the machine, we can say that allocations that are not multiples of 2 will not be advantageous, and are not worth exploring. This naturally leads to a simple strategy: if the system employs simultaneous multithreading, then restrict the system to selecting allocations that are multiples ("chunks") of the number of threads per core.

Machine	SMT?	Chunk size restriction
tilegx	No	1
openlab08	No	1
tempo-mic0	4-way	4
triad	No	1
bhindi	2-way	2

Table 6.3: Heuristically-chosen chunk sizes used for each machine

We implemented this heuristic in our implementation of `scafd`. The chunk sizes used for each machine can be found in Table 6.3. (A chunk size of 1 indicates that any allocation size is allowed.)

6.2.2 Rate-limiting to Reduce Overheads

Because the SCAF client runtime works by adding extra logic and instrumentation between parallel sections, we must be sure to avoid allowing this overhead to become excessive. Serial experiments execute only once per section, but a process must evaluate its parallel efficiency throughout execution. Reading hardware counters, timers, and computing efficiency estimates can cause a quickly-iterating parallel process to significantly increase the amount of time spent between parallel sections. If this overhead grows to be excessive, then any benefits from improved allocations may be negated. We discovered that the Xeon Phi and Tile-GX in particular were sensitive to this overhead, likely due to their simple cores and otherwise good scalability. The average cost for an E5 Xeon core to read hardware counters is less than 1000 cycles. By comparison, the average cost to read hardware counters on a Xeon Phi thread is about 3600 cycles, and the average cost to read hardware counters on a

Tile-GX core is about 5000 cycles. When combined with the facts that the Xeon Phi and Tile-GX cores are additionally more parallel and clocked significantly slower, the cost of instrumentation becomes relatively high.

As a result, we independently rate-limit (i.e., lower the frequency of) two periodic events in the `libscaf` client library: reading hardware counters for instrumenting parallel sections, and communication with the central SCAF daemon. This rate-limiting is implemented using a token-bucket technique, which works as follows. Each client maintains "allowance" variables for instrumentation and communication which are evaluated every time the client would communicate with the SCAF daemon or read hardware counters before a parallel section. The wall time since the last evaluation is used to increment allowances according to the maximum allowed rate. If the allowance then has a value of at least one, the rate-limited action is allowed, and the allowance is decreased by one. If the allowance is less than one, then the action is skipped. We store the most recent IPC measurement for each parallel section so that it may be re-used in the event that a section's instrumentation is skipped. The effect of communicating with the SCAF daemon less often is that new allocation decisions will be delayed until the next communication is allowed.

As an example, consider the UA benchmark on the Xeon Phi. UA is a benchmark which iterates through parallel sections on the order of a thousand times per second. In order to evaluate the overhead due to communication with `scafd` and instrumentation, we disabled serial experiments and ran UA by itself, in a non-multiprogrammed scenario. Using an unmodified OpenMP runtime, UA achieves a speedup of about 75.6X over one thread. With a non-rate-limited `libscaf`, this

drops to less than less than 48X – lower by a factor of more than 1.5X. Such a large slowdown cannot be overcome in a multiprogrammed scenario, and would result in poor system efficiency. With communications with `scafd` limited to 5 Hz and the number of sections instrumented per second limited to 32 Hz, the slowdown due to SCAF's instrumentation and communication is reduced to less than one half of one percent. This per-process overhead is small enough to accept as the cost necessary to improve overall system efficiency. To be clear, because there is no multiprogramming in this test SCAF is not expected to ever provide any improvement; it can only match the unmodified (non-SCAF) OpenMP runtime.

6.2.3 Virtual Memory Management

On Tile-GX and Xeon Phi we encountered two significant difficulties related to virtual memory management. SCAF uses `fork()` in a high-performance context which is probably not frequently tested by Linux kernel developers. Serial experiments depend on `fork()` to copy the process's state just before a parallel section's first execution begins. This should ideally be done quickly and should have no lasting effects on the main process. However, on the Xeon Phi we found that `fork()` could be unusually slow, and on the Tile-GX we found that the main process's performance was severely impacted after `for()` had completed.

Xeon Phi/Linux 2.6.38 When we first ported SCAF to the Xeon Phi, we discovered that the `fork()` system call itself was prohibitively slow when forking a multithreaded process. We found that the Phi cores have only 8 interrupt vectors for

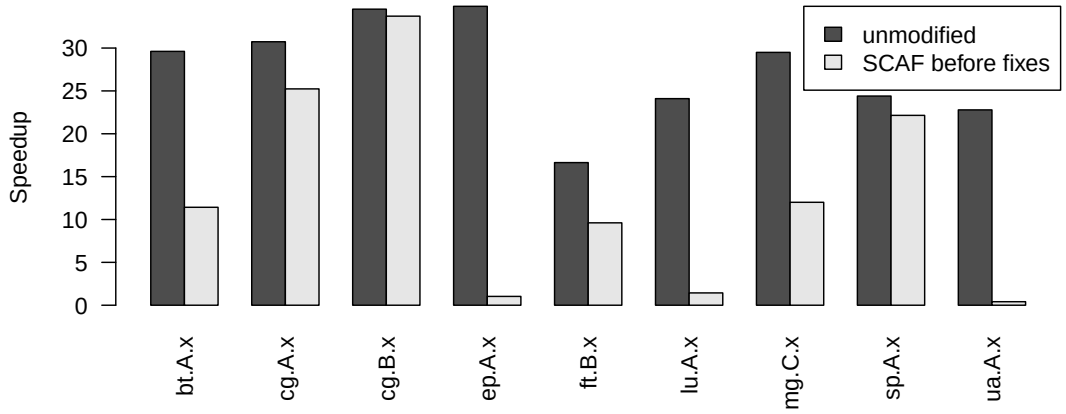


Figure 6.5: Tile-GX: Slowdowns with unmodified virtual management due to `fork()`

TLB invalidations, and that `fork()`'s time was mostly spent doing TLB shootdowns. The Phi's Intel-maintained Linux 2.6.38 kernel project was using an outdated TLB shootdown strategy, so we have backported a series of patches contributed by Intel to mainline Linux (3.6+) in 2012 which improved TLB shootdown performance. The resulting kernel has acceptable `fork()` system call performance, in line with our other platforms. These findings are summarized in Table 6.2.

Tile-GX/Linux 3.4 When we first ported SCAF to Tile-GX, we discovered that our benchmarks were catastrophically slowed down after serial experiments ran, as seen in Figure 6.5. The culprit turned out to be some Tile-GX-specific heuristics within the Linux kernel's memory management.

The Tile-GX is unique in that it exposes a virtual cache subsystem to the Linux kernel and user space. Each page of memory can be marked as either "homed" at a specific tile, or cached in a distributed L3 cache, using a hash function ("hash-for-home") to determine a home tile for each cache line in the page. The hope is that L2 misses can be fulfilled by another tile via the L3 cache rather than main memory.

The Tilera Linux kernel has various heuristics which it uses to automatically decide how to cache pages. By default, stack pages are homed local to the tile on which a software thread is executing, while heap and data pages are cached using hash-for-home. Upon a `fork`, the Linux kernel has to set up a copy-on-write (COW) copy of the process's address space. This includes the individual stacks for each of the threads in the process. Setting up the COW mapping does not allocate any new physical pages or change any homing attributes. There are now two problematic scenarios:

1. If the child process terminates without writing to a COW page, the kernel "breaks" the COW for that page, removing the read-only mapping for the child and marking the parent's copy as read-write again. At this point, the original heuristics re-home the page to the particular tile that called the `fork`. In our case, where a multithreaded program is being forked, this has the effect of re-homing all of the process's threads to a single thread.

2. If the parent (that is, any thread in the process) does write to a COW page, then Linux must make a physical copy of the page. In this case, it decides to home the page again at the particular tile that called the `fork`. This again has the effect of re-homing all of the process's threads to a single thread.

In these cases, performance drops massively: the threads no longer have stacks homed locally, and the `fork`ing thread's tile struggles to satisfy cache requests from all of the other threads. These two scenarios are not specific to SCAF, but the use of `fork` to clone a multithreaded process's address space for a child process is

uncommon.

We modified the VMM code in Linux specific to Tile-GX in order to improve the heuristics in the case of forking a multithreaded process. Specifically, a page should not be re-homed on COW-break or COW-copy if it is a stack page for another thread in the process. We found that these heuristic changes eliminated the slowdowns we had previously observed. Table 6.2 summarizes our experience with SCAF and Tile-GX's kernel-exposed cache subsystem.

6.2.4 "Lazy" Experiments

The last optimization we have applied to our scheme is a policy of "lazy" experiments. The objective of this policy is to avoid any overhead associated with experiments until we are actually in a multiprogrammed scenario. SCAF clients receive the total number of active clients from `scafd`, and simply defer experiments while they are the only client. Another benefit of this scheme is that it can allow clients to avoid running experiments for setup-oriented parallel sections, which often only run once. The overhead incurred by serial experiments when running multiprogrammed scenarios is generally justified by the benefits of improved allocations. With lazy experimenting in non-multiprogrammed scenarios, there is no overhead due to serial experiments because they are never needed.

Chapter 7

Evaluation

In this section, we offer results which demonstrate the advantages of deploying SCAF on a multiprogrammed system. We compare three configurations: 1) the system's unmodified OpenMP implementation, 2) simple equipartitioning, and 3) the fully dynamic SCAF implementation, as described in this paper. In practice, the unmodified configuration is by far the most common since it requires no setup and is readily available. The second configuration, equipartitioning, represents the state of the art which does not require a priori testing. However, we are not aware of any widely-available OpenMP runtimes which implement equipartitioning. SCAF is the configuration presented in this paper, which also needs no a priori profiling.

7.1 Multiprogramming with NPB Benchmark Pairs

For each platform and configuration we evaluated all 36 pairs of 9 NPB benchmarks. The NAS Parallel Benchmarks (NPB benchmarks) [9], developed by the NASA Advanced Supercomputing Division, is a collection of parallel programs primarily drawn from the field of computational fluid dynamics. They are widely used to test the performance of parallel computers, from workstations to supercomputers. In this paper, we evaluate our system with nine of ten benchmarks in the OpenMP version of NPB. The 'DC' benchmark is omitted because it is storage oriented, and

several of our platforms (Xeon Phi and Tile-GX) have no local storage. The remaining 9 benchmarks are included: 'IS', which is an example of a non-malleable process supported as described in section 5.3; 'EP', which consists primarily of a single long-running parallel section supported as described in section 5.4; 'LU'; 'BT'; 'MG'; 'FT'; 'UA'; 'SP'; and 'CG'. In each multiprogramming pair, program 1 first begins execution, and then after 10 seconds program 2 begins execution. This series of events could easily occur when two users are interactively using a remote machine, launching processes while unaware of one another. The 10 second delay was introduced in order to avoid unrealistic precisely-simultaneous starts, but our results are not dependent on this delay. Problem sizes for each benchmark were chosen such that solo runtimes would be as similar as possible, with a size of 'T' indicating a custom problem size we created out of necessity. The average benchmark runtime is roughly one to three minutes, depending on the platform.

Figure 7.1 shows the distribution of SCAF's improvement factors over equipartitioning across all benchmark pairs for all 5 machines. The pairs are sorted (per machine) in order of their improvement factors, so that the distribution is clear. Examining Figure 7.1, we can see that with the exception of "tempo-mic0," the Xeon Phi, the majority of pair scenarios see either no significant harm or a benefit from SCAF over equipartitioning. Overall, the Xeon Phi does not tend to benefit from SCAF vs simple equipartitioning. As a result, we recommend that SCAF be used only it its limited equipartitioning-only mode on the Xeon Phi. (In this mode, serial experiments are not necessary.) Figure 7.2, similarly, shows the distribution of SCAF's improvement factors over an unmodified system for all 5 machines. Exam-

ining Figure 7.2, we see that the overwhelming majority of scenarios benefit from SCAF by significant (1.1X - 1.5X) factors. These significant improvements are often due to the conversion of fine-grained time sharing to space sharing. (I.e., by avoiding oversubscription.) The interesting exception is the UltraSparc T2, "triad," on which fine-grained time sharing seems to incur a smaller penalty. Unfortunately, triad manages this using commercial Solaris thread scheduler and synchronization implementations, the sources for which are unavailable. Please note that pair names are not shown in Figures 7.1 or 7.2 because each pair has a different rank on each of the machines when sorted by its improvement factor.

Figure 7.3 shows all pairwise results on the Tile-GX-based "tilegx." Here, SCAF is significantly faster than the unmodified runtime in all cases except the (bt.A,is.T) pair, where the sum of speedups about breaks even (improvement factor = 0.99X). When compared to equipartitioning, SCAF is faster or about equal in all but 4 pairs. In the 23 pairs where SCAF is significantly faster than equipartiting, the mean improvement is a factor of 1.18X. Overall, SCAF does very well on the Tile-GX due to its lack of hardware multithreading and predictable scalability.

Figure 7.4 shows all pairwise results on our 2-way Xeon E5-2690 "openlab08" machine. SCAF shows significant improvement of about 1.18X in 17 pairs, and breaks even in about 13 pairs. SCAF is generally faster than the unmodified configuration; the average improvement is about a factor of 1.4X. Here SCAF is able to make an improvement because this system uses no hardware multithreading, and scales in a predictable manner.

Figure 7.5 shows all pairwise results on our Xeon Phi "tempo-mic0," excluding

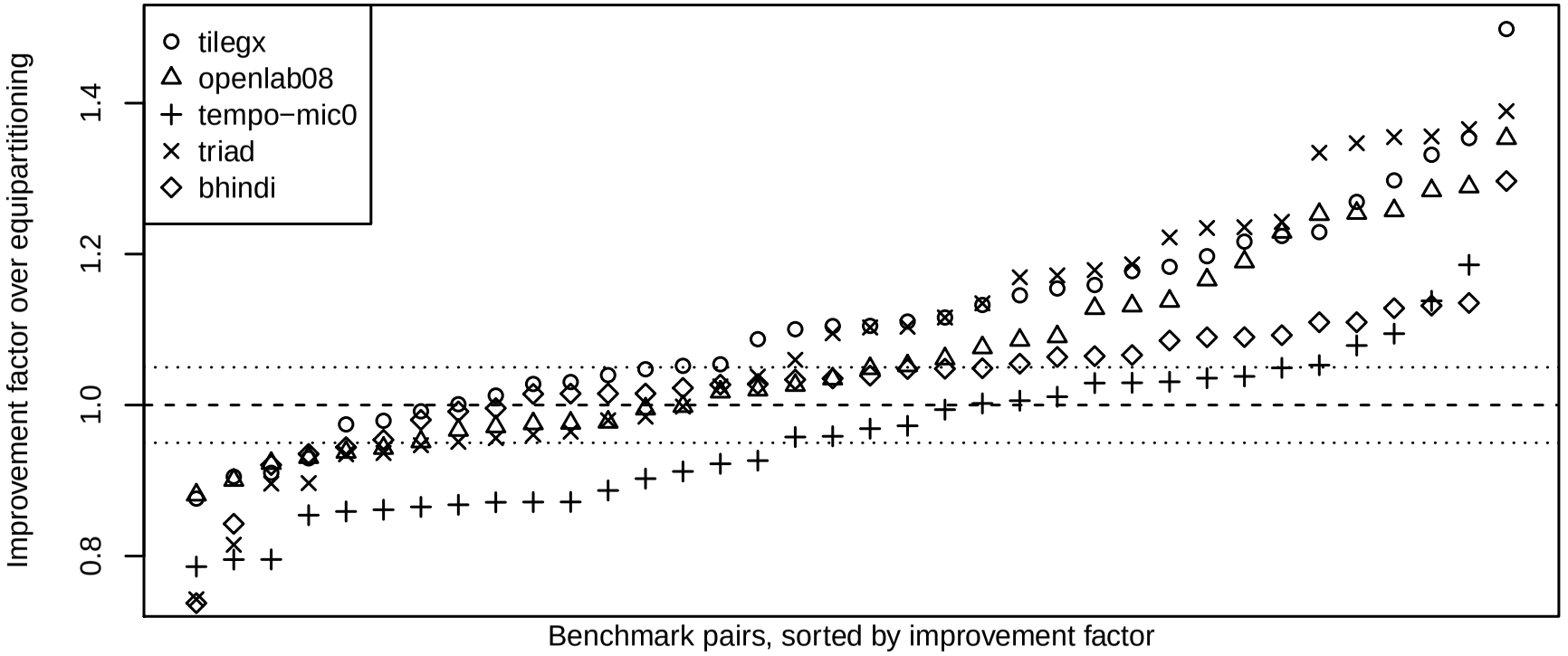


Figure 7.1: Distribution of improvement over equipartitioning on all machines. 4 out of 5 machines perform well. Only "tempo-mic0" does not show an improvement, so we recommend that SCAF not be used for that and similar machines for which our test suite performs poorly.

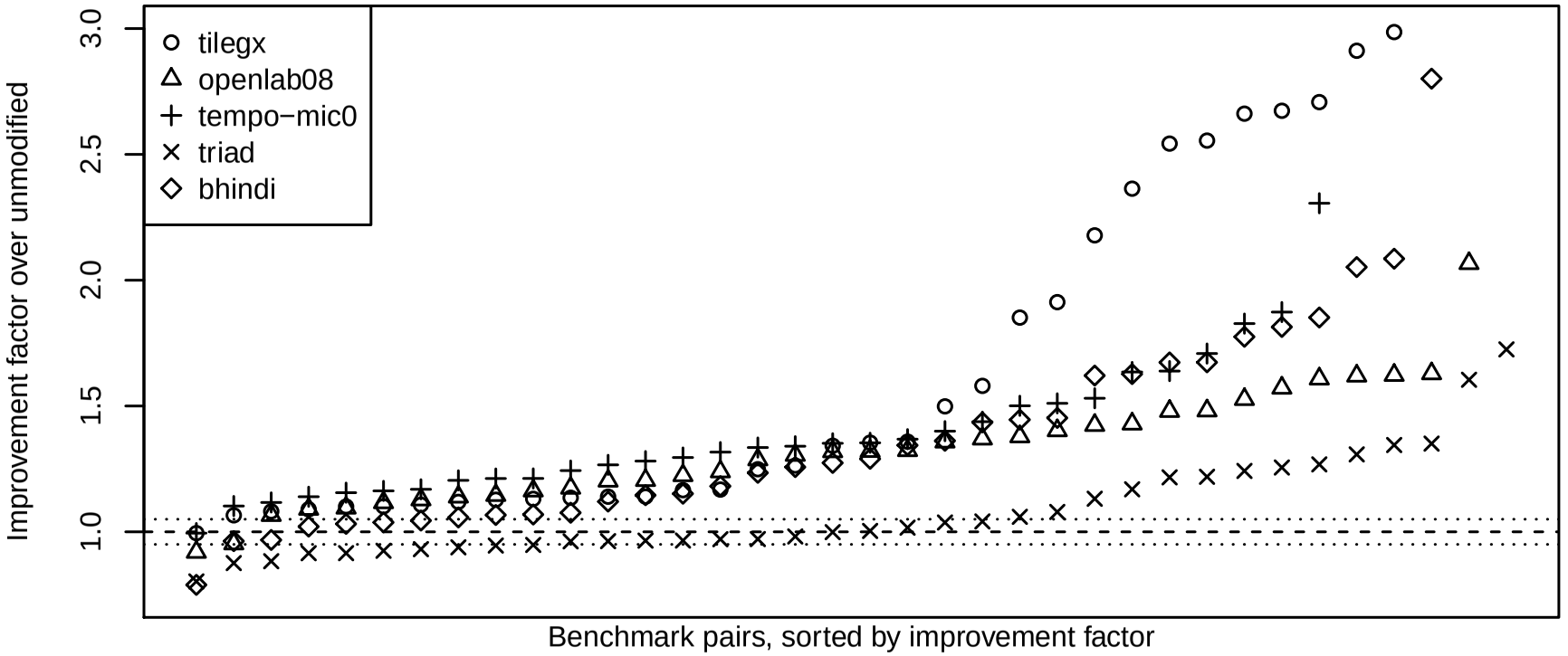


Figure 7.2: Distribution of improvement over an unmodified system on all machines.

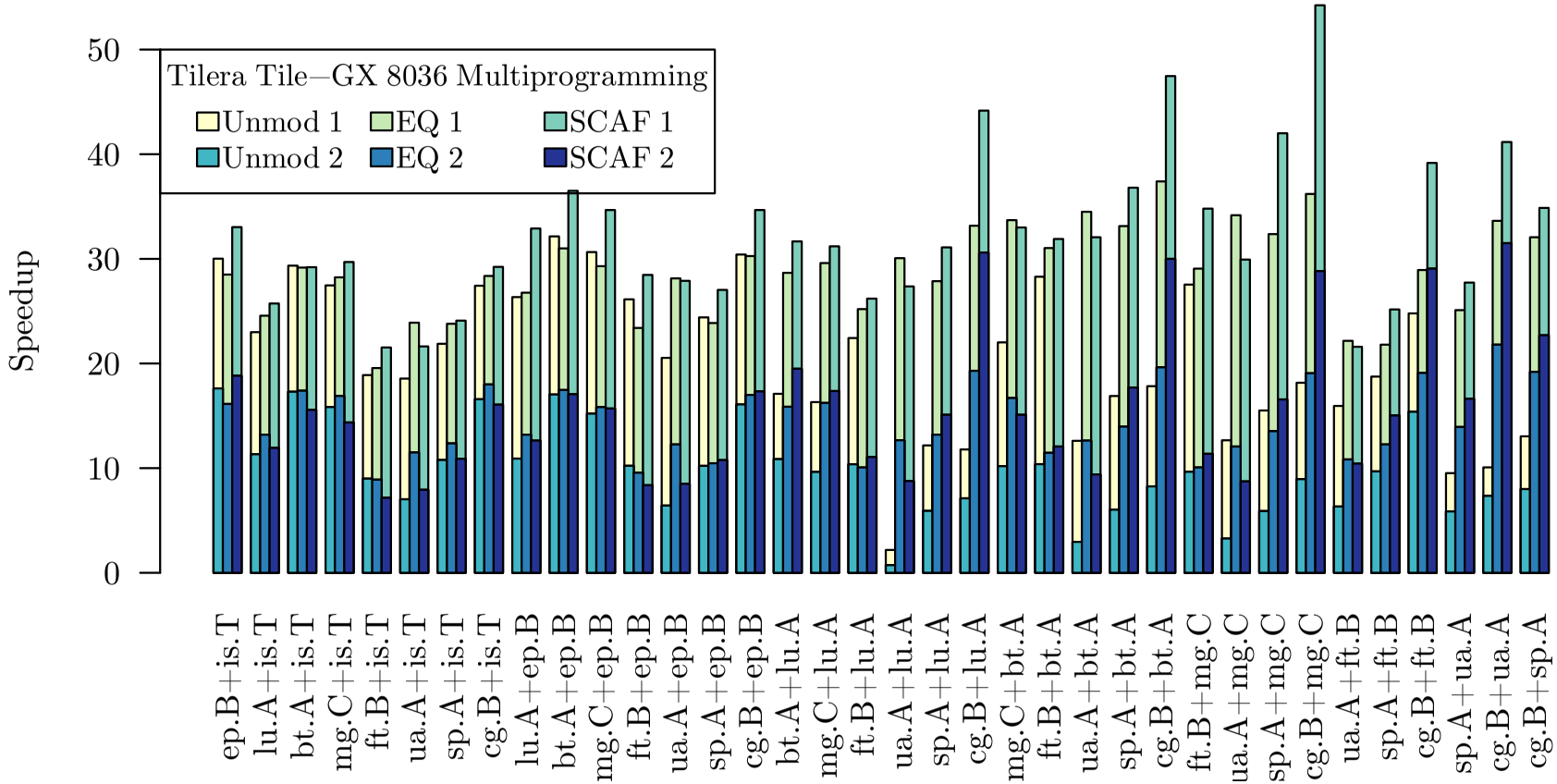


Figure 7.3: Results on a Tilera Tile-GX with 36 hardware contexts

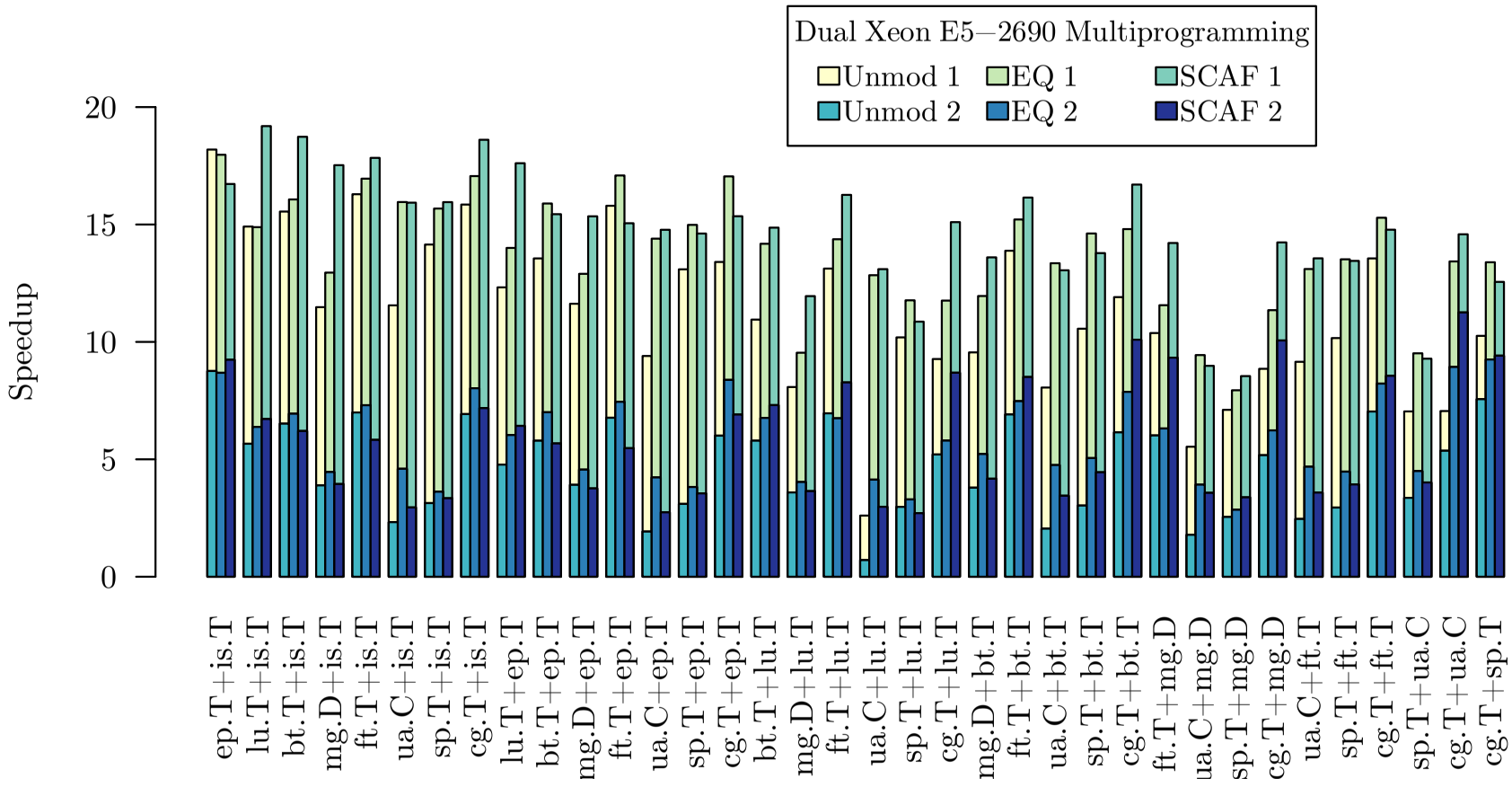


Figure 7.4: Results on a dual Xeon E5-2690 with 16 hardware contexts

the (FT.U,MG.U) pair due to a lack of main memory. (Smaller data sets would result in runtimes that are too small.) This is the only machine for which we generally see no significant improvement or loss from SCAF when compared to equipartitioning. This is due to the fact that that all but one of the benchmarks scale very similarly, with good efficiency until around half of the hardware contexts are used, resulting in a relatively low potential for SCAF to make improvement. SCAF is still generally significantly faster than an unmodified system, yielding an average improvement by a factor of 1.72X. As a result, we don't recommend the full SCAF scheme on the Xeon Phi; equipartitioning is generally better. The SCAF distribution will include special recognition for the Phi, and recommend using only equipartitioning. Additionally, the distribution will include an automated test suite based on the NAS benchmarks which can be used to identify any other systems where SCAF is unhelpful. For such machines, we recommend SCAF should be turned off. Once SCAF becomes open-source software, we hope that the open-source community will maintain a list of machines for which SCAF is not helpful.

Figure 7.6 shows pairwise results on "triad,", our UltraSparc T2. The T2's results are interesting because the unmodified system, with heavy oversubscription, is actually competitive with equipartitioning. (We suspect this is due in part to the scheduler in Solaris 11.2, although its implementation is not available for examination.) Still, because the T2 uses primarily a form of temporal hardware multithreading, its speedup behavior is predictable for SCAF. The SCAF system still improves on equipartitioning by an average factor of about 1.22X in 20 pairs, and on the unmodified system by an average factor of 1.27X in 14 pairs, and was the best choice

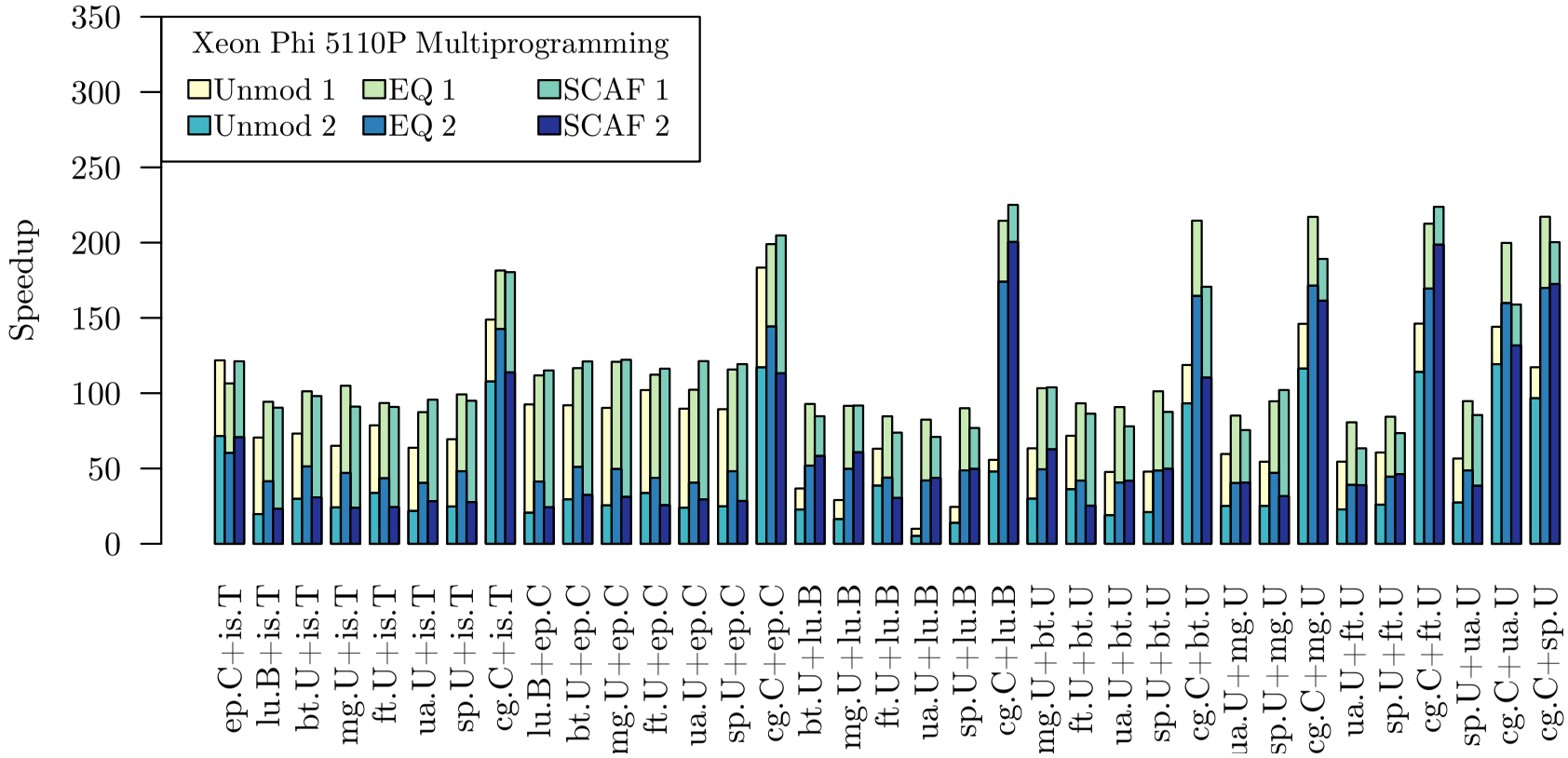


Figure 7.5: Results on a Xeon Phi 5110P with 240 hardware contexts

in the most pairs.

Figure 7.7 shows all pairwise results on "bhindi," our 2-way Opteron 6212 machine. Although SCAF is faster than equipartitioning in 14 of the pairs by an average of 1.11X, it only breaks even for 17 pairs. As discussed in section 6.1.1, the use of a form of simultaneous multithreading results in complex speedup behavior which can't always be successfully reasoned about with SCAF's simple speedup model.

Table 7.1 summarizes the results shown in Figures 7.1 through 7.7. The achieved improvement factor in the sum-speedup of each pair compared to equipartitioning and an unmodified system is considered on a pair-by-pair basis. We consider improvement factors deviating from 1.0X by more than 0.05X in either direction to be significant. An improvement factor of 1.05X or greater is classified as a significant "win," an improvement factor less than 1.05X but greater than 0.95X as "breaking even," and an improvement factor below 0.95X as a loss. For each machine, we provide the percentage of pairs falling in to each category and the mean improvement factor within each category when compared to both equipartitioning and an unmodified system.

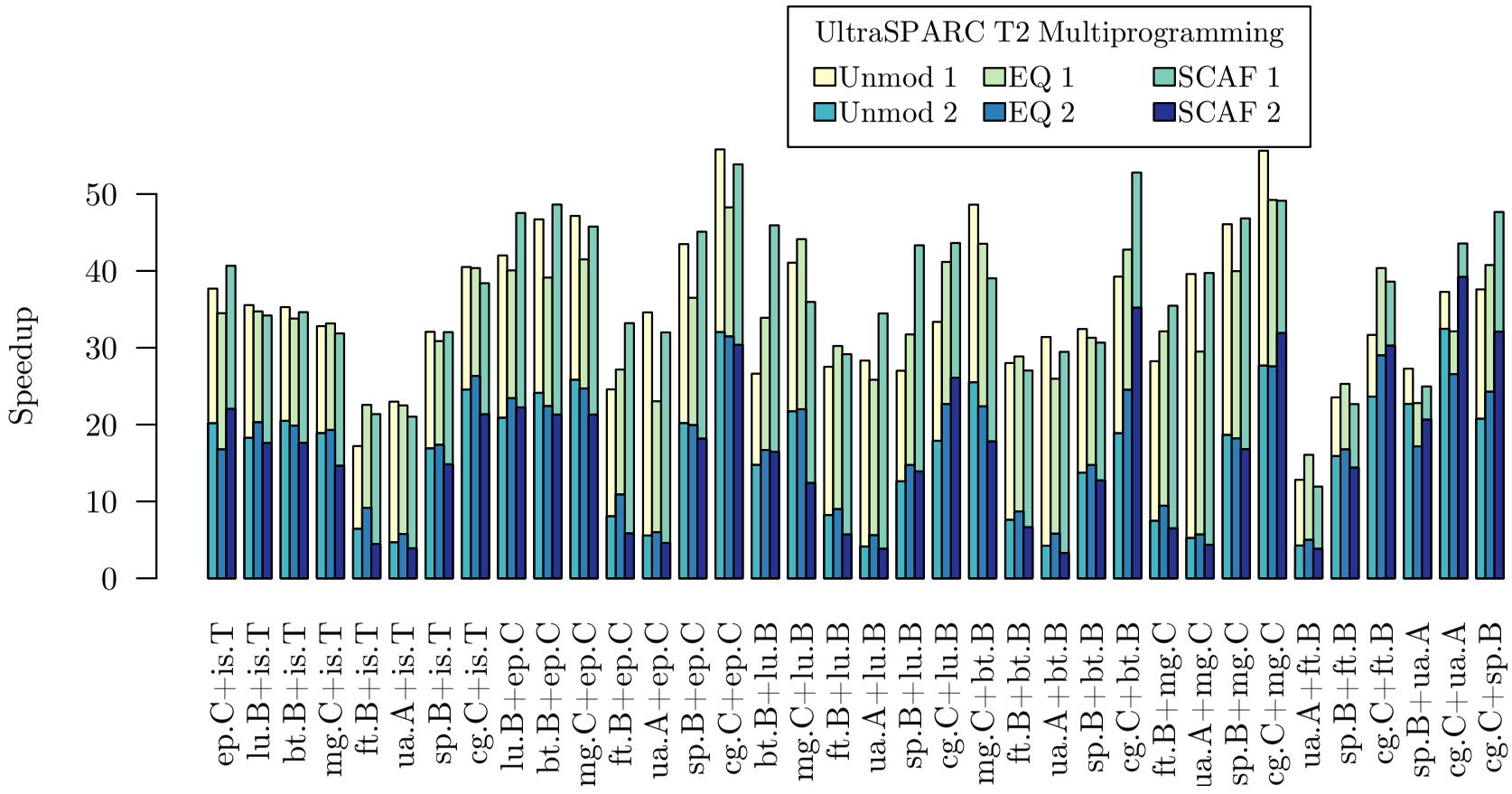


Figure 7.6: Results on an UltraSparc T2 with 64 hardware contexts

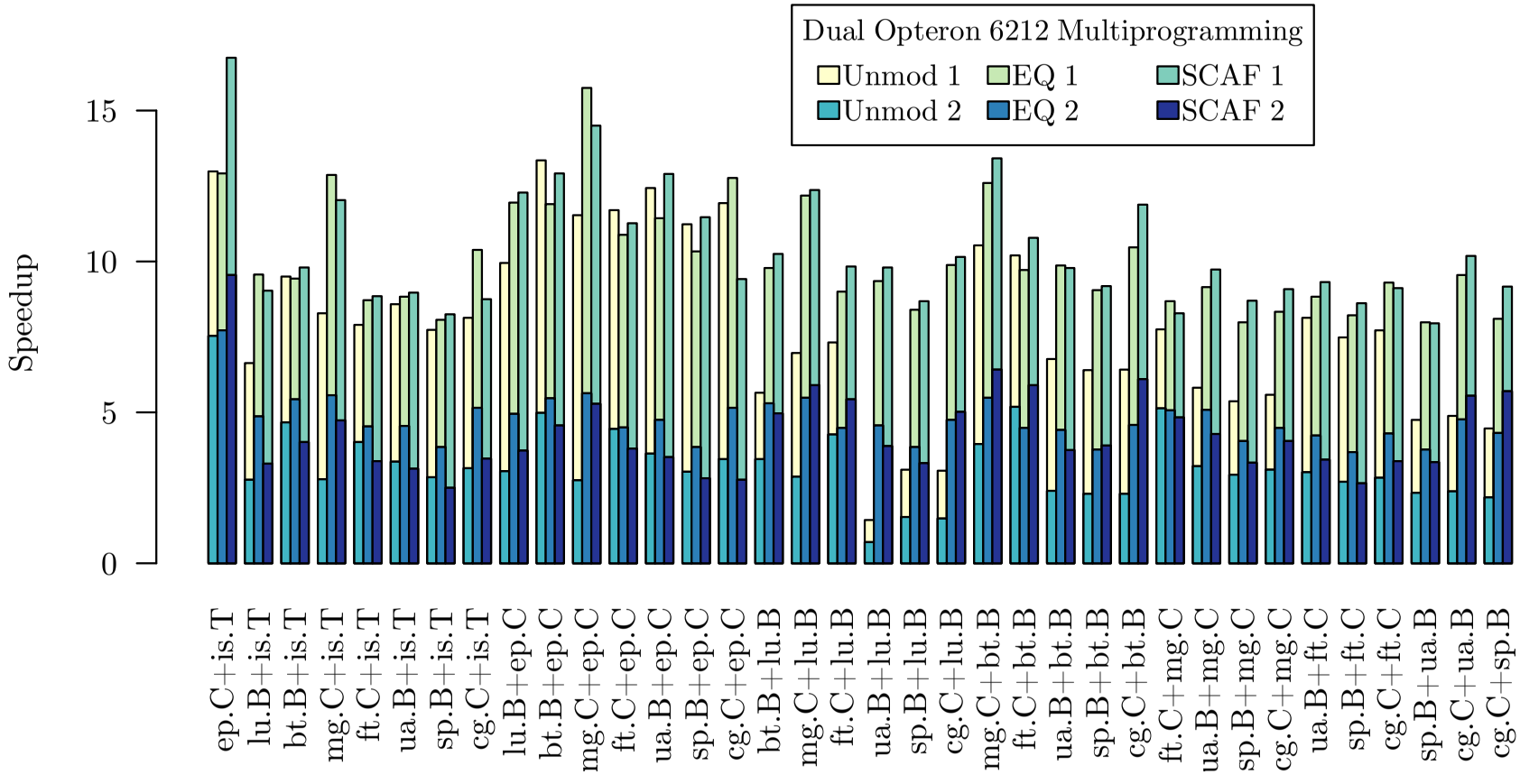


Figure 7.7: Results on a dual Opteron 6212 with 16 hardware contexts

Machine	Wins v. EQ	Wins v. Unmod	Even v. EQ	Even v. Unmod	Loss v. EQ	Loss v. Unmod
tilegx	1.18X, 64%	1.72X, 97%	1.01X, 25%	1.00X, 3%	0.91X, 11%	N/A, 0%
openlab08	1.18X, 47%	1.39X, 94%	1.00X, 36%	0.96X, 3%	0.92X, 17%	0.92X, 3%
tempo-mic0	1.11X, 14%	1.57X, 97%	1.00X, 40%	1.00X, 3%	0.87X, 46%	N/A, 0%
triad	1.22X, 56%	1.27X, 39%	0.98X, 25%	0.99X, 33%	0.87X, 19%	0.91X, 28%
bhindi	1.11X, 39%	1.56X, 81%	1.01X, 47%	1.01X, 17%	0.87X, 14%	0.79X, 3%

Table 7.1: Summary of mean pairwise results. Only "tempo-mic0" does not perform well.

7.2 Detailed 3-Way Multiprogramming Scenario

The previous section (7.1) focuses on 2-way multiprogramming, but SCAF can handle greater numbers of clients. The severity of oversubscription seen in the unmodified configuration only increases with increased multiprogramming. In this section, we focus on a particular 3-way multiprogramming experiment on the UltraSparc T2. Running all 3-way scenarios would result in too many combinations, so we provide a detailed look at only one in order to provide insight into the mechanics of SCAF. When run by itself, the first benchmark, cg.C, scales extremely well on a T2, with a maximum speedup near 50X on 64 threads. The other two benchmarks, sp.B and lu.B, scale more modestly, with maximum speedups of 25-30X on 64 threads.

In this experiment, cg.C begins at time 0, then sp.B 10 seconds later, and lu.B after an additional 10 seconds. We executed this workload for each of the three configurations: an unmodified system, equipartitioning, and SCAF. Table 7.2 summarizes the results, while Figure 7.8 plots log output of the SCAF daemon (`scafd`) after running this scenario.

In Table 7.2, we see that the unmodified configuration manages a mediocre sum speedup of 31.5X due to severe oversubscription. Each benchmark uses 64 threads. As the third benchmark begins, the SunOS scheduler is left to timeshare 192 CPU-bound threads to 64 hardware contexts. This severe oversubscription results in lackluster performance from all three benchmarks due to context switching and hardware contention.

The equipartitioning configuration manages to avoid oversubscription and as

Configuration	Process	Runtime	Speedup	$\sum$ Speedup
Unmodified	CG	435.9s	13.1X	31.5X
	SP	474.6s	9.6X	
	LU	507.3s	8.8X	
Equi-partitioning	CG	374.0s	15.5X	40.7X
	SP	380.8s	12.2X	
	LU	349.8s	13.0X	
SCAF	CG	172.2s	35.7X	59.3X
	SP	374.0s	12.5X	
	LU	424.0s	11.1X	

Table 7.2: Summary of the 3-way multiprogramming scenario

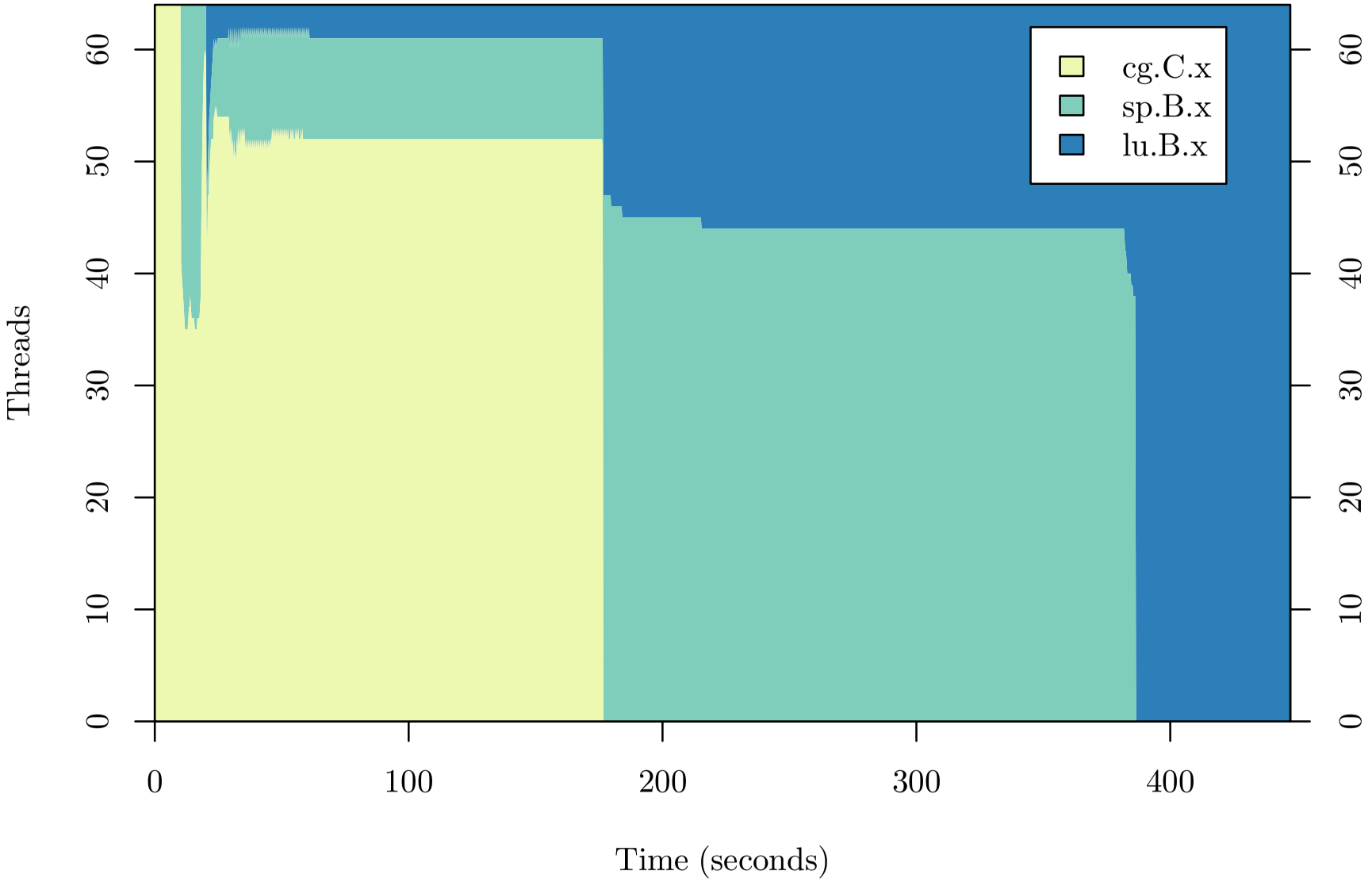


Figure 7.8: SCAF behavior during a 3-way multiprogramming example

a result completes all three benchmarks faster. At first, cg.C runs on all 64 threads. After 10 seconds, sp.B begins and is given 32 of cg.C's threads. Finally, after lu.B begins, sp.B and lu.B are allocated 21 threads while cg.C is allocated 22 threads. lu.B finishes first, leaving cg.C and sp.B each 32 threads. Finally, cg.C finishes, leaving sp.B to complete with all 64 threads. Here, the sum speedup increases to 40.7X due to a lack of oversubscription.

With SCAF, we see improved overall performance. Figure 7.8 shows SCAF's allocations throughout the experiment. Initially, cg.C runs on all 64 threads. After 10 seconds, sp.B begins and is briefly allocated approximately 29 of the threads. At 20 seconds, sp.B begins as well. Very quickly, SCAF's begins to observe that cg.C is scaling particularly well, resulting in a significant allocation of 52 threads. Due to this large allocation, cg.C finishes after only 172.2 seconds, after which sp.B and lu.B are allowed to expand to use the full 64 contexts. At this point, sp.B and lu.B are observed to have similar performance, with sp.B having shown better results while cg.C was running. As a result, SCAF allocates about 44 threads to sp.B and the remaining 20 go to lu.B. Finally, at 384 seconds sp.B finishes and lu.B is left to complete on all 64 threads. The achieved sum speedup is 59.3X, an improvement by factors of 1.88X and 1.45X over the unmodified and equipartitioning configurations, respectively.

7.3 Oracle Comparison

In this section, we present experimental results designed to compare SCAF's partitioning decisions with true optimal partitionings. In other words, if we had an "oracle" which could advise `scafd` on the current optimal allocation, how does SCAF's behavior differ? Four multiprogramming pair cases are presented, illustrating that SCAF makes reasonable decisions.

In general, even given the advantage of offline analysis and limiting ourselves to 2-way multiprogramming, determining a true oracle's response is difficult. This is because the search space is quite large: for every point in multiprogrammed time, all possible allocations must be considered. For example, on a 36-core machine and a 200-second multiprogrammed region with our default 4Hz resolution, the number of possible dynamic allocation plans is about $(36-1)^{200 \cdot 4}$. Thus, naively testing even just one benchmark pair on one test platform would require infeasibly many trials.

However, many of the NAS benchmarks we have chosen for evaluation exhibit a steady state of speedup behavior, allowing us to reasonably compare SCAF against a simplified oracle which chooses only a single optimal partitioning for the entire multiprogrammed period. In other words, the oracle simply chooses the best static partitioning. To discover the best partitionings, we performed extensive offline analysis of two benchmark pairs on two different machines: the TileGX and the dual Opteron 6212. The TileGX was chosen as an example of a platform where SCAF performs well, and the Opteron machine was chosen as an example of a platform where SCAF provides less improvement. Speedups presented in this section (7.3)

are based on more precise *partial* benchmark runtimes in order to isolate behavior within the multiprogrammed region, as opposed to the speedups presented in section 7.1, which are based on whole-program runtimes.

7.3.1 TileGX

We have chosen two benchmark pairs to evaluate on TileGX: SP+LU and CG+FT. These two pairs were chosen as examples of pairs with small and large differences in scalability, respectively. Figure 7.9 shows the information the oracle would have at its disposal when multiprogramming CG and FT. This figure represents the individual and sum speedups obtained during the multiprogrammed time period when varying the static partitioning from 1 vs 35 threads to 35 vs 1 threads. Here, we see that CG and FT have comparable overall scalability, although individual speedups have interesting plateaus and then steps at 21 and 31 threads. As a result, an oracle would choose an extreme allocation such as 5 cores for SP and 31 for LU. However, because SCAF does not implement a search of all possible allocations, it tends to settle on a more moderate split around 18 cores for each of SP and LU. In this case, the sum speedup for SCAF is 25X, which is only 6% worse than the oracle's 26.7X speedup. The reason for this is clear: `scafd` begins with equipartitioning, and since the speedups are flat in this area, `scafd` sees no benefit in moving the split in either direction. The result is that SCAF's partitioning is also more fair than the Oracle's in this case. The oracle makes no attempt at fairness, whereas SCAF attempts to prevent starvation of any thread; however that is not the reason for the

more equitable allocation of SCAF in this case.

In contrast, consider Figure 7.10, which shows the information that an oracle would have while multiprogramming SP and LU. In this case, CG scales far better than FT, and the oracle would choose the partitioning that gives CG 35 cores and FT only 1. SCAF jumps from equipartitioning to a similar partitioning of about 30 vs 6. The speedup for SCAF in this case is about 25.6X, about 9% worse than the oracle's 28X speedup at an extreme partitioning. The uneven partitioning is driven by the observed speedup advantage of CG, but it is not as extreme as the oracle's partitioning due to the logarithmic speedup model which SCAF uses to reason about speedups. In general, the use of a logarithmic family of speedup models strongly discourages extreme partitionings; this is effectively a heuristic in SCAF working toward fairness.

7.3.2 Dual Opteron 6212

On the dual Opteron 6212 machine, we chose benchmark pairs LU+BT and FT+LU. These two pairs were again selected as examples of pairs with small and large relative differences in scalability, respectively. Figure 7.11 depicts the information an oracle would have at its disposal when multiprogramming LU and BT. Here, LU and BT have similar speedup behavior, with the maximum sum of speedups found at a split of 10 vs 6 cores. In practice, this is also the allocation which SCAF chooses most often. In other words, both the oracle and SCAF acheive a sum speedup of 10.1X in this scenario. SCAF arrives at this allocation based on the slight advantage

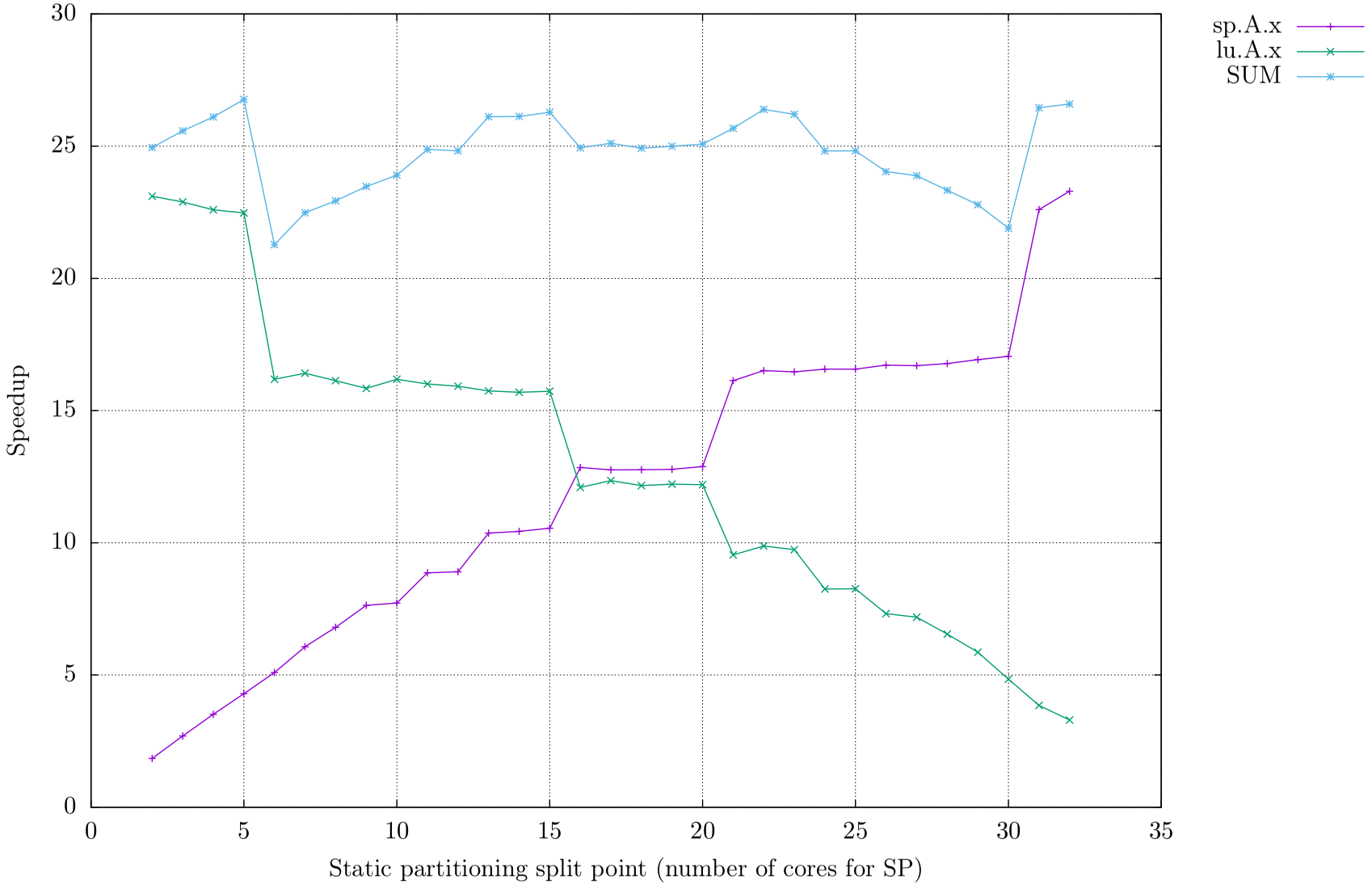


Figure 7.9: Exploration of all static partitionings of SP+LU on Tile-GX. The two intersecting lines represent SP and LU's individual speedups, while the upper line represents their sum of speedups.

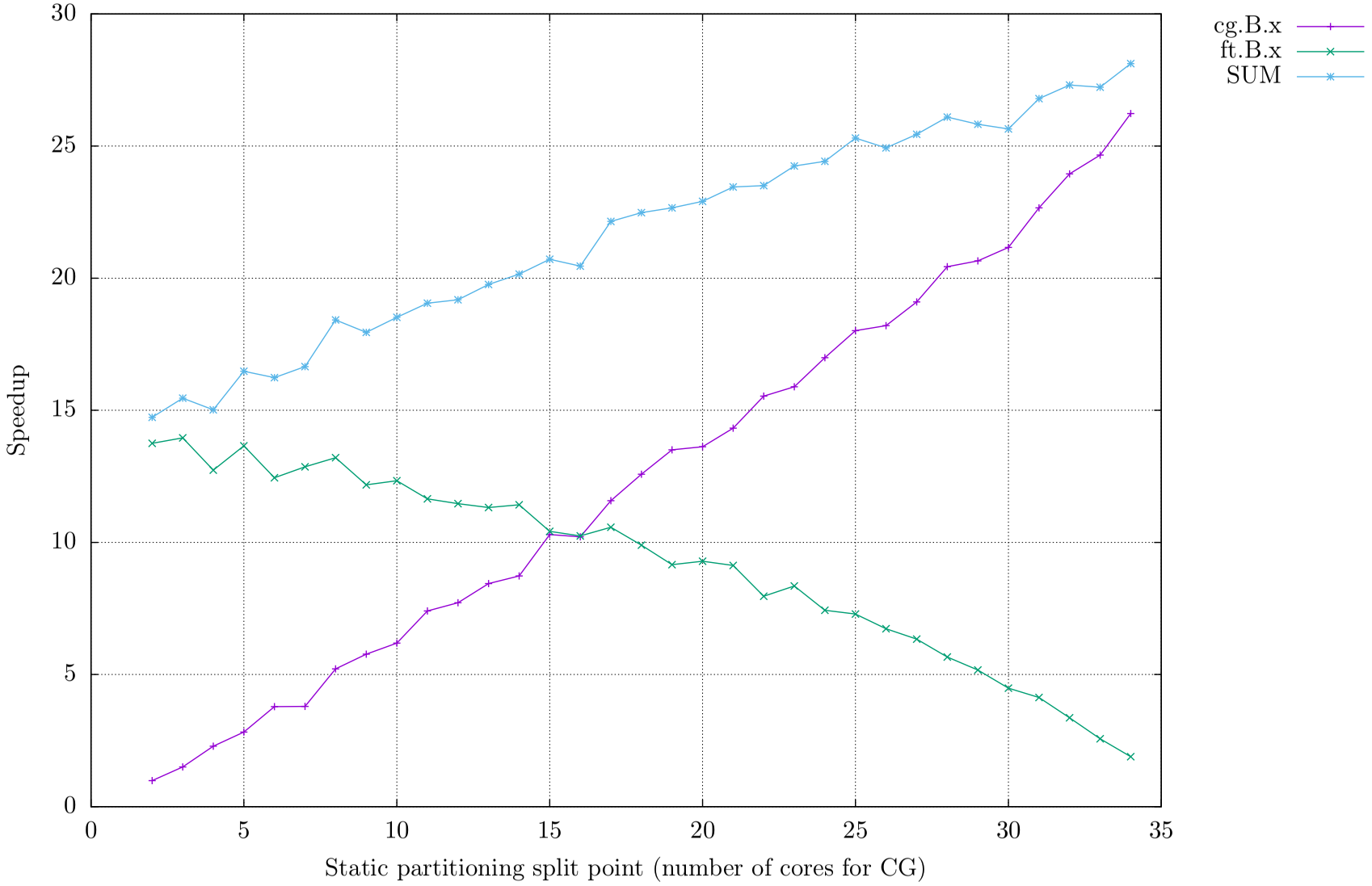


Figure 7.10: Exploration of all static partitionings of CG+FT on Tile-GX. The two intersecting lines represent CG and FT's individual speedups, while the upper line represents their sum of speedups.

of LU's efficiency. Note that SCAF will actually never choose the worse-performing splits of 9 vs 7 nor 11 vs 5 due to the "chunk" size of 2 used on machines with 2-way simultaneous multithreading, so the 10 vs 6 split is essentially adjacent to equipartitioning.

In contrast, Figure 7.12 shows the information available to an oracle when multiprogramming FT and LU. Here, FT exhibits poor scaling when compared to LU. Additionally, we see that speedup behavior for LU is complex for high (9-15) core counts while FT is also running. As a result, an oracle can see that a partitioning of 3 threads for FT and 13 for LU is actually optimal. However, since SCAF reasons using simple logarithmic speedup functions and only explores allocations that are multiples of two, it most often chooses an allocation of 6 threads for FT. However, sometimes it chooses 4 or 8 due to the inconsistent results from LU. In other words, SCAF correctly chooses in favor of the better-performing benchmark, but complicated speedup behaviors due to SMT can prevent it from arriving at the true optimal partitioning. In the common case where SCAF chooses 6 threads for FT, it achieves a sum speedup of 10.3X, only 3% lower than the oracle's 10.6X sum speedup.

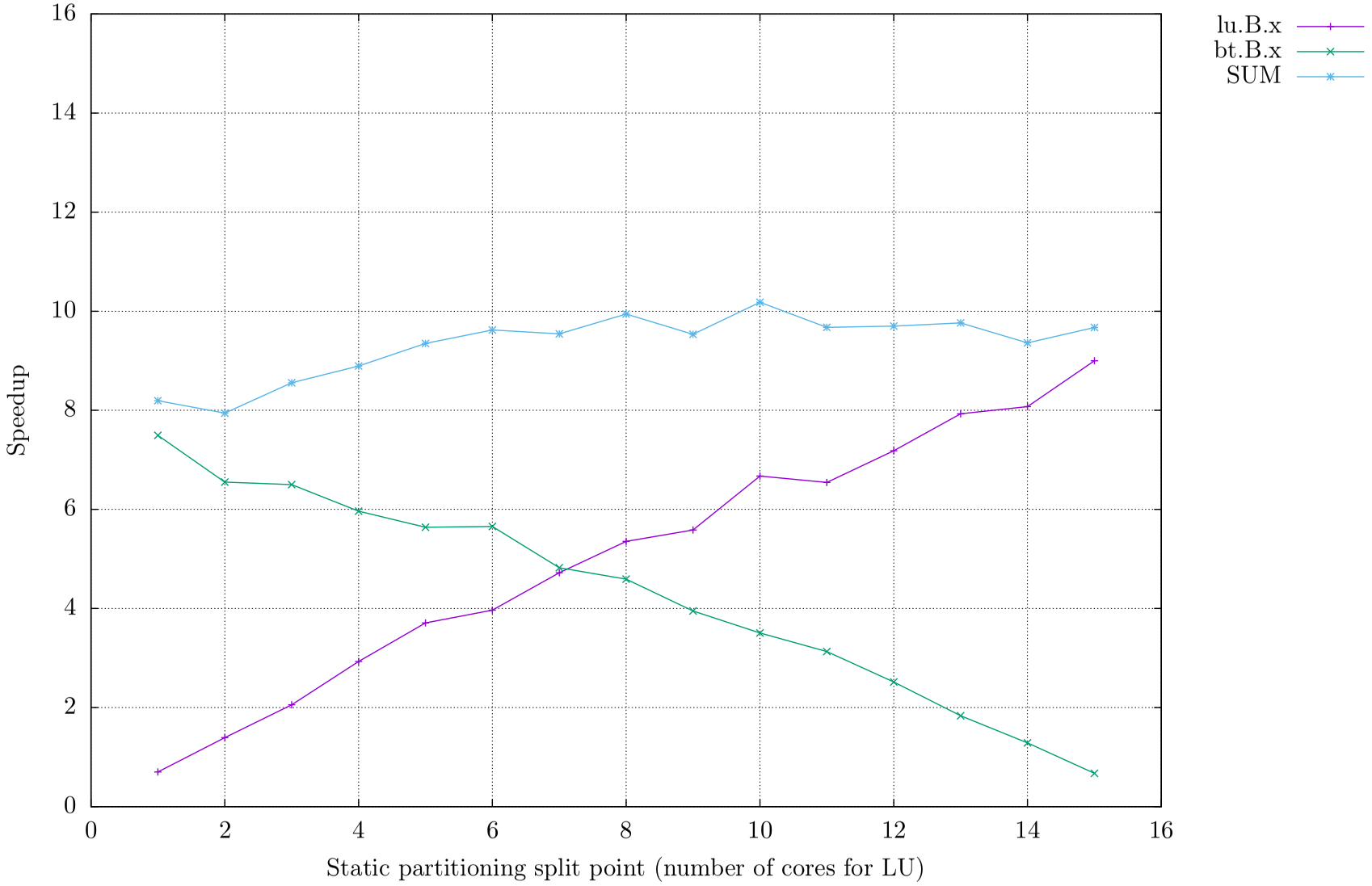


Figure 7.11: Exploration of all static partitionings of LU+BT on a dual Opteron 6212 machine. The two intersecting lines represent LU's and BT's individual speedups, while the upper line represents their sum of speedups.

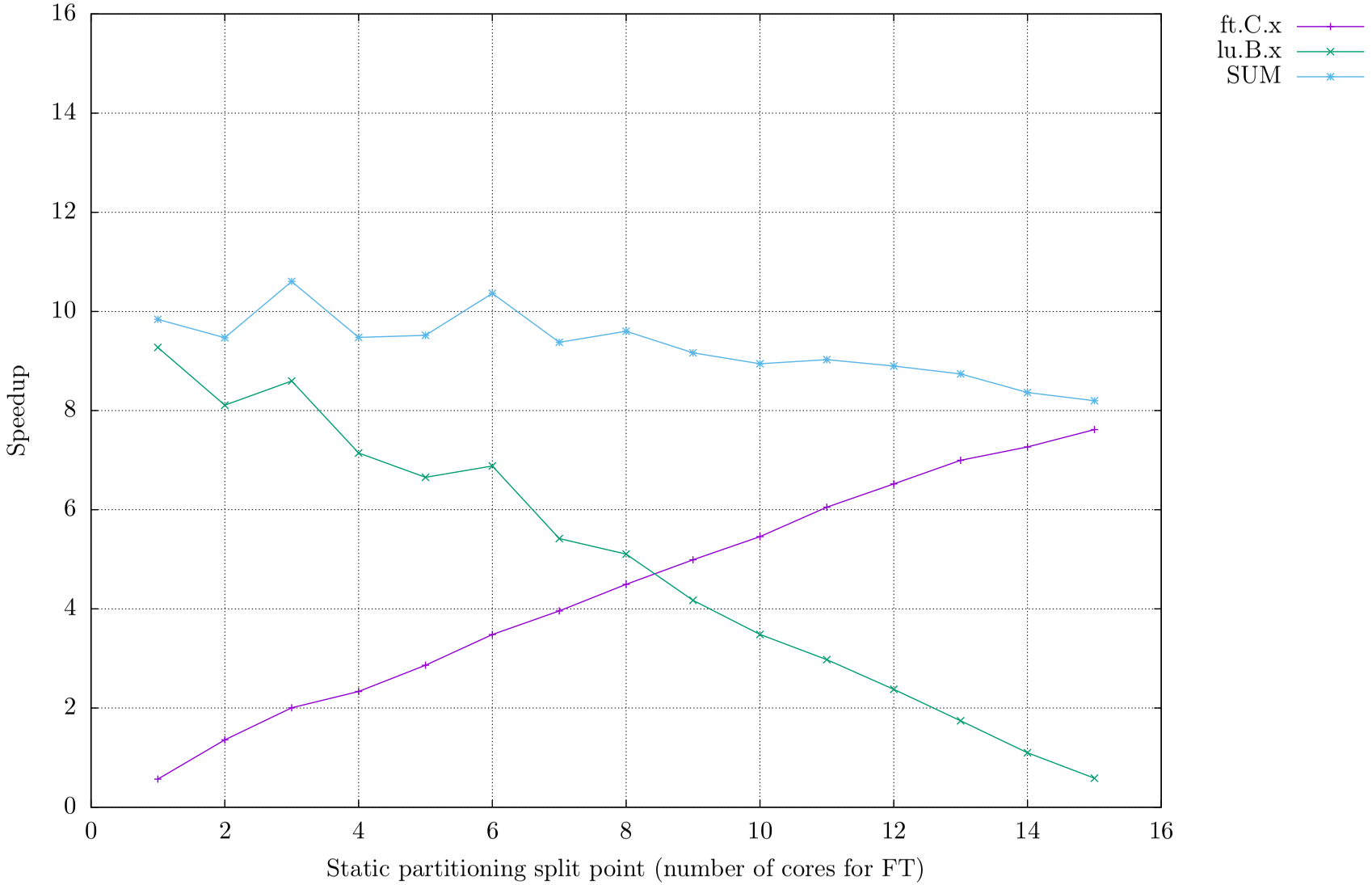


Figure 7.12: Exploration of all static partitionings of FT+LU on a dual Opteron 6212 machine. The two intersecting lines represent FT's and LU's individual speedups, while the upper line represents their sum of speedups.

Chapter 8

Future Extensions to SCAF

Future directions for the practical development of SCAF that the community could investigate included the following. With this work and the planned open-source release of SCAF, with hope to catalyze such future work in the open-source community.

8.1 Porting additional runtime systems

SCAF may be useful in combination with additional runtime systems beyond GNU OpenMP, such as Open64's OpenMP runtimes and Intel's TBB library. Although most of the runtime changes will be very similar, some differences will arise for TBB. TBB makes use of a dynamic work-stealing model, which may result in design changes when modifying TBB to support changing the number of threads used at runtime and require additional methods for estimating parallel efficiency. However, OpenMP is the de-facto standard for shared memory programs today, and is far more prevalent.

8.2 Expanding results to additional hardware platforms

SCAF has been primarily tested on Linux with Intel x86_64 SMPs. SCAF may prove useful if ported to more platforms, such as SGI UV 2000 NUMA systems with

512-1024 cores, IBM Power systems, or virtual machines with virtualized hardware counters. SCAF and its techniques are intended to become more useful and effective on more parallel systems such as these.

8.3 Periodic lightweight experiments

One advantage of having a method for collecting serial IPC at runtime is that it allows the measurement to be repeated periodically or on certain triggers. Some long-running processes may have serial IPC which is very dependent upon input. In these cases where inputs can vary greatly, it may not even be possible to gather comprehensive information on serial performance even we are able to run tests ahead of time. A SCAF system which can re-run serial experiments could overcome these difficulties.

8.4 Supporting applications at the thread level

Currently, SCAF's mechanisms for controlling parallelism (changing the number of threads at the start of parallel sections and setting process affinities) are simple and not the focus of the work. These mechanisms also restrict supported programs to those based on supported parallel runtime libraries. It may be worthwhile to investigate if it is possible to extend SCAF to support controlling parallelism in a POSIX threads library. We expect such an effort would be highly nontrivial without requiring recompilation, akin to implementing a new user-space threading library which is backwards-compatible with current kernel-managed libraries. However, if

such a mechanism could be devised without requiring recompilation, SCAF could transparently support a wider array of programs.

8.5 Resource allocation toward power efficiency

Our current work always seeks to maximize efficiency with respect to system performance. However, the resulting decisions may be power inefficient. With new techniques, it may be possible to allocate such that hardware threads are intelligently left unused in order to target power optimization as an additional goal. These techniques may also be applicable to non-multiprogrammed scenarios.

As a motivating example, consider a system running only a single badly-scaling multithreaded processes on an 8-core machine. Say running this process on 2 threads achieves a 2x speedup, but only consumes energy at a rate of 20 watts. Running it on 8 threads may improve speedup to 2.1x, but at a cost of 80 watts. In this scenario, a very reasonable policy might be to optimize for power consumption rather than performance, and decide to run only 2 threads.

8.6 Linux scheduler improvements for groups of tasks

The Linux scheduler schedules only individual software threads, and discards any notion of processes or groups of threads. Although this information (e.g., "threads 4-8 constitute process 200") is available to the kernel, it is not used when scheduling threads to cores in order to minimize the computational requirements of the thread scheduler.

As a result, in practice high-performance applications are often run by users with explicit directions to the thread scheduler about how to distribute the threads of a its process. The threads' processor affinities are manually specified, based on the assumption that the Linux scheduler cannot be expected to come up with an equivalent plan. Ideally, users should not have to do such things for optimal performance.

We may investigate opportunities for improving the Linux scheduler by taking into account special knowledge of groups of threads, such as processes, without imposing excessive overhead.

8.7 Resource allocation across virtual machines

We may investigate efficient resource allocation of virtual machines, possibly within or across VM hosts. Virtual machines can themselves been seen as multi-threaded processes, similar to OpenMP programs. However, virtual machines are currently generally allocated in fairly static manners, and there may be room for improvement by way of a more dynamic solution.

8.8 Automatic software-based heartbeats

SCAF uses hardware counters in order to reason about performance efficiency, but this is only because it is the most generic metric available. Unfortunately, for various reasons, hardware counters are not always available. (For example, within most virtual machines.) Additionally, depending on the workload, hardware-counter-based metrics such as IPC can be highly misleading. We may investigate a technique

for injecting software-based heartbeats into as a generic alternative to hardware counters, useful for reasoning about effective runtime performance of applications.

Chapter 9

Conclusion

This work has shown that for multiprogrammed workloads neither a priori testing, nor simple equipartitioning is generally satisfactory. We argue that none of the related work has caught on in practice, due to the significant inconvenience to the user of performing profiling or testing ahead of time, or because they require changes to the program or re-compilation. We have presented a drop-in system, SCAF, which includes a technique for collecting equivalent information at runtime, paying only a modest performance fee and enabling sophisticated resource management without recompilation, modification, or profiling of programs. We believe that such resource management will be important as hardware becomes increasingly parallel, and as more parallel applications become available.

Appendix: Appendix

A.1 Detailed Hardware Topologies

Figures A.1-A.5 show detailed topologies for each of the tested platforms. All figures were generated by "hwloc" [70].

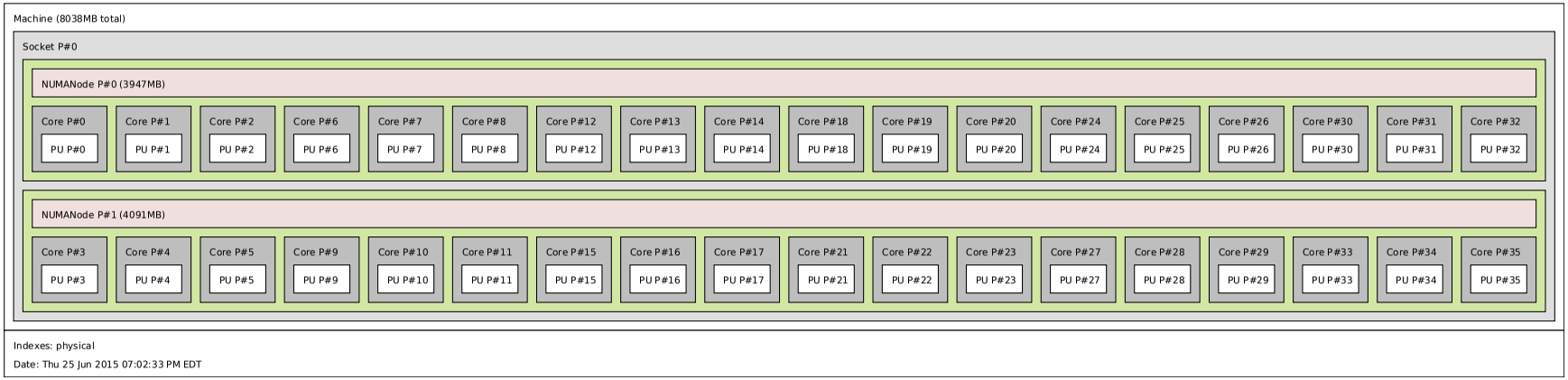


Figure A.1: Hardware Topology of a 36-core TileGX

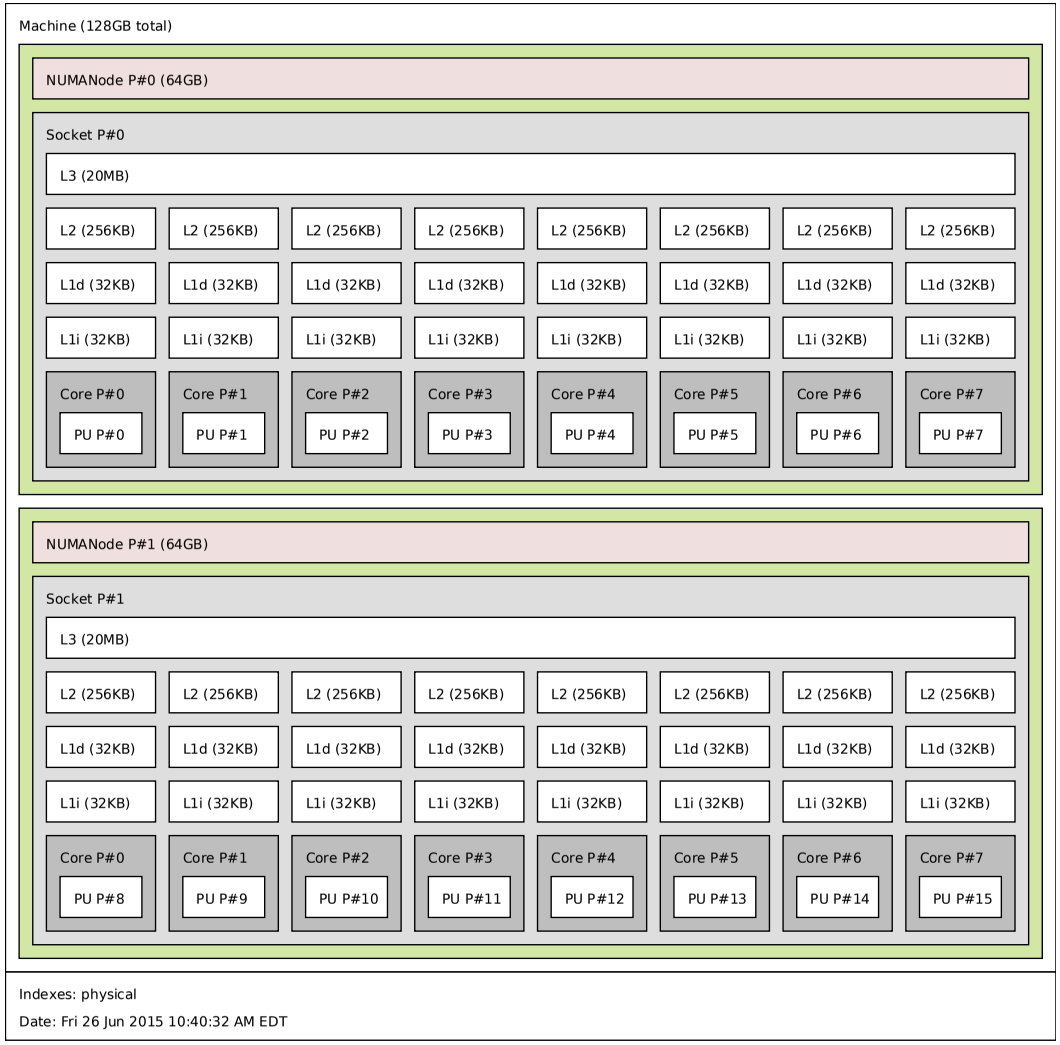


Figure A.2: Hardware Topology of a 16-core Xeon E5-2690

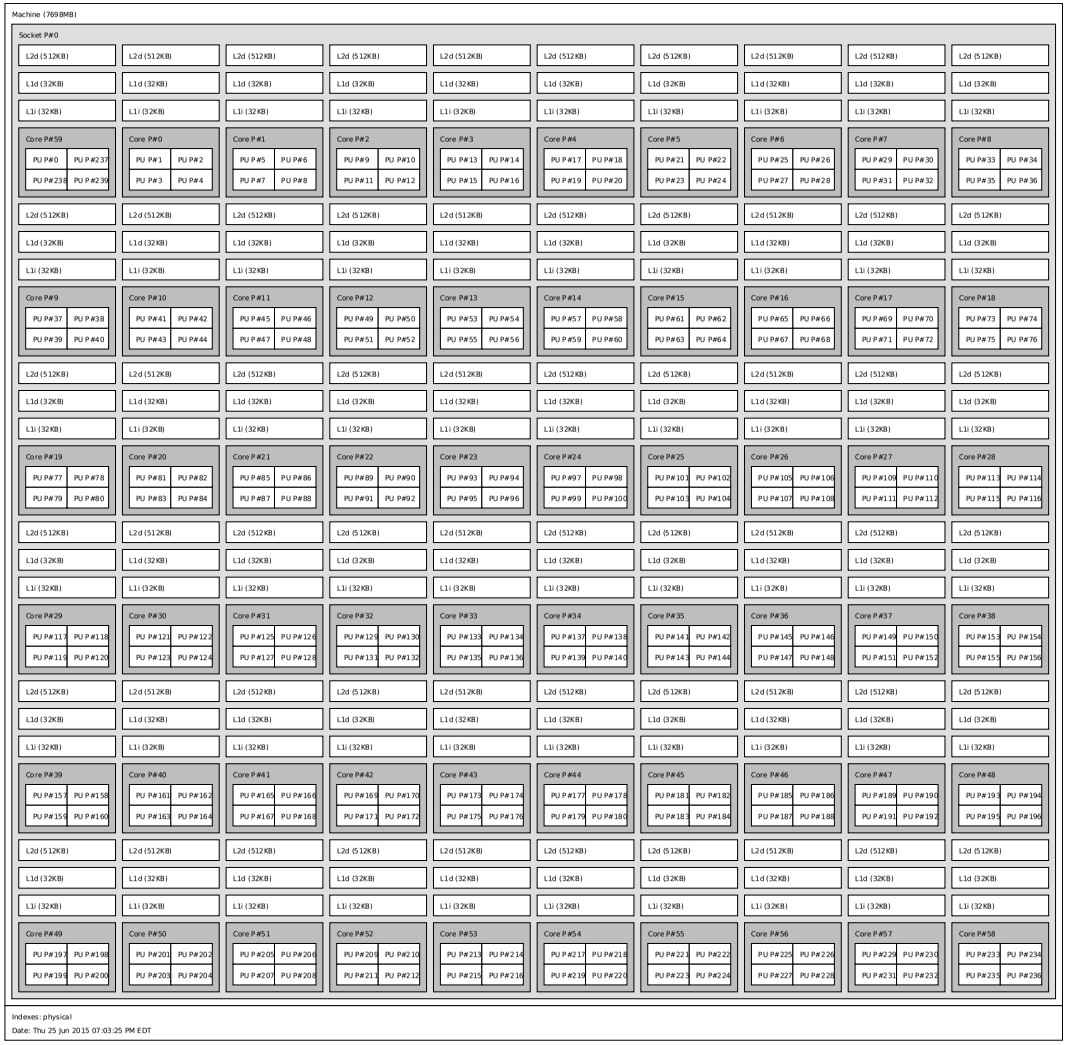


Figure A.3: Hardware Topology of a 60-core Xeon Phi 5100p

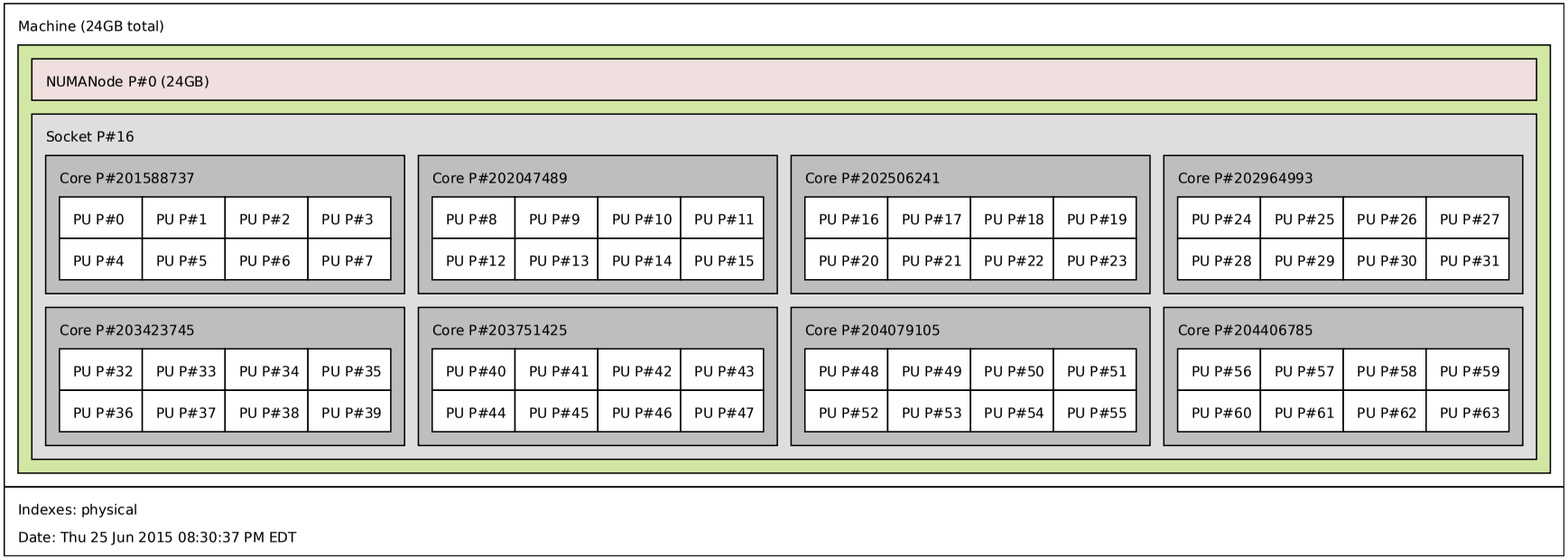


Figure A.4: Hardware Topology of an 8-core UltraSparc T2

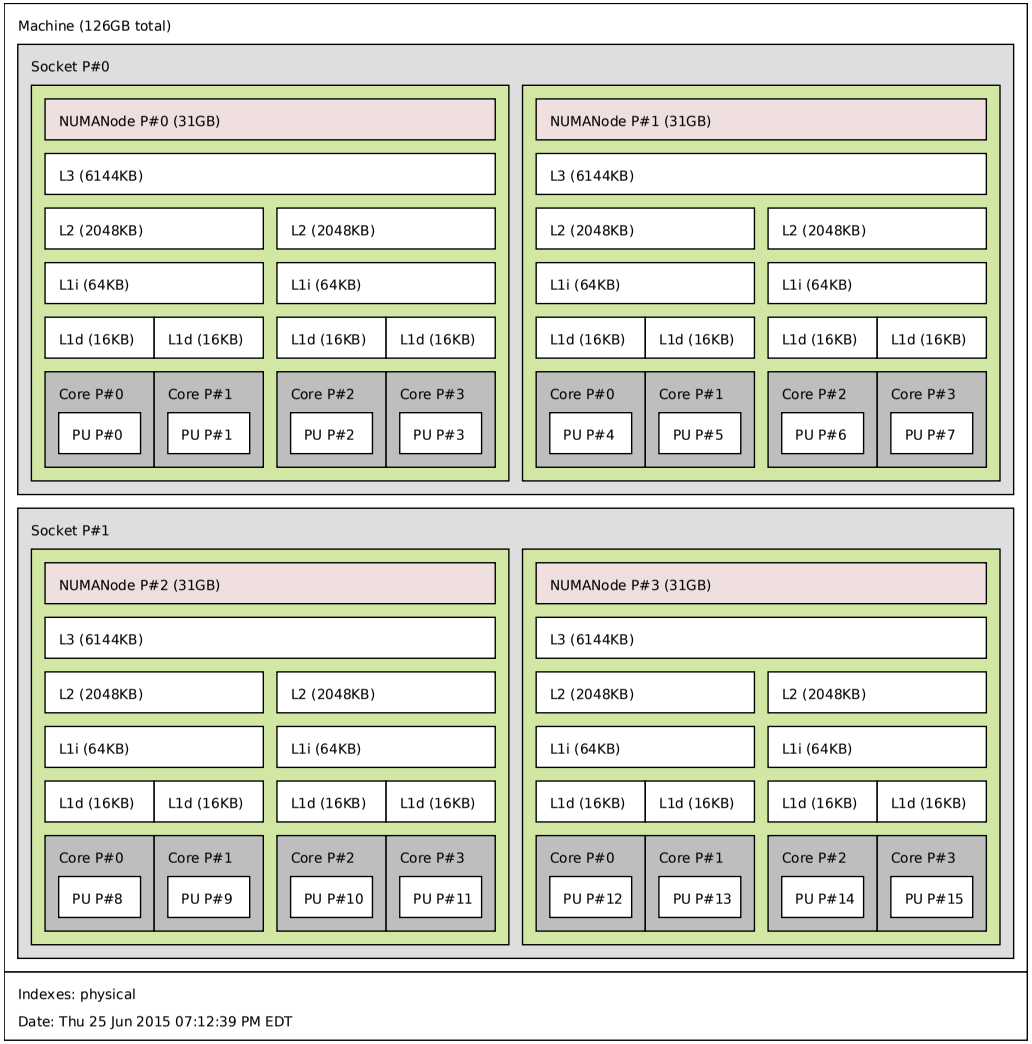


Figure A.5: Hardware Topology of an 8-core Opteron 6212

A.2 Using IPC ratios to estimate true speedup on TileGX

Figures A.6-A.14 support our claims that IPC ratios (rather than raw IPC) can be used reasonably to estimate true speedups. In these figures, whole-program speedup is plotted in a solid line, while raw IPC (total, for all cores) is plotted using a dashed line. The points plot an IPC-based *estimate* of speedup, which is simply the total IPC divided by the process's known single-threaded IPC. IPC was reported using the Linux `perf stat` tool, while speedups are as reported by the NAS benchmarks.

To aid the reader in comparing speedups, all plots are shown on the same vertical and horizontal axes. The important thing to note from these graphs is that while raw IPC is a poor indicator of scalability, IPC can still be used to reasonably estimate speedup if we compare it to a known serial speedup.

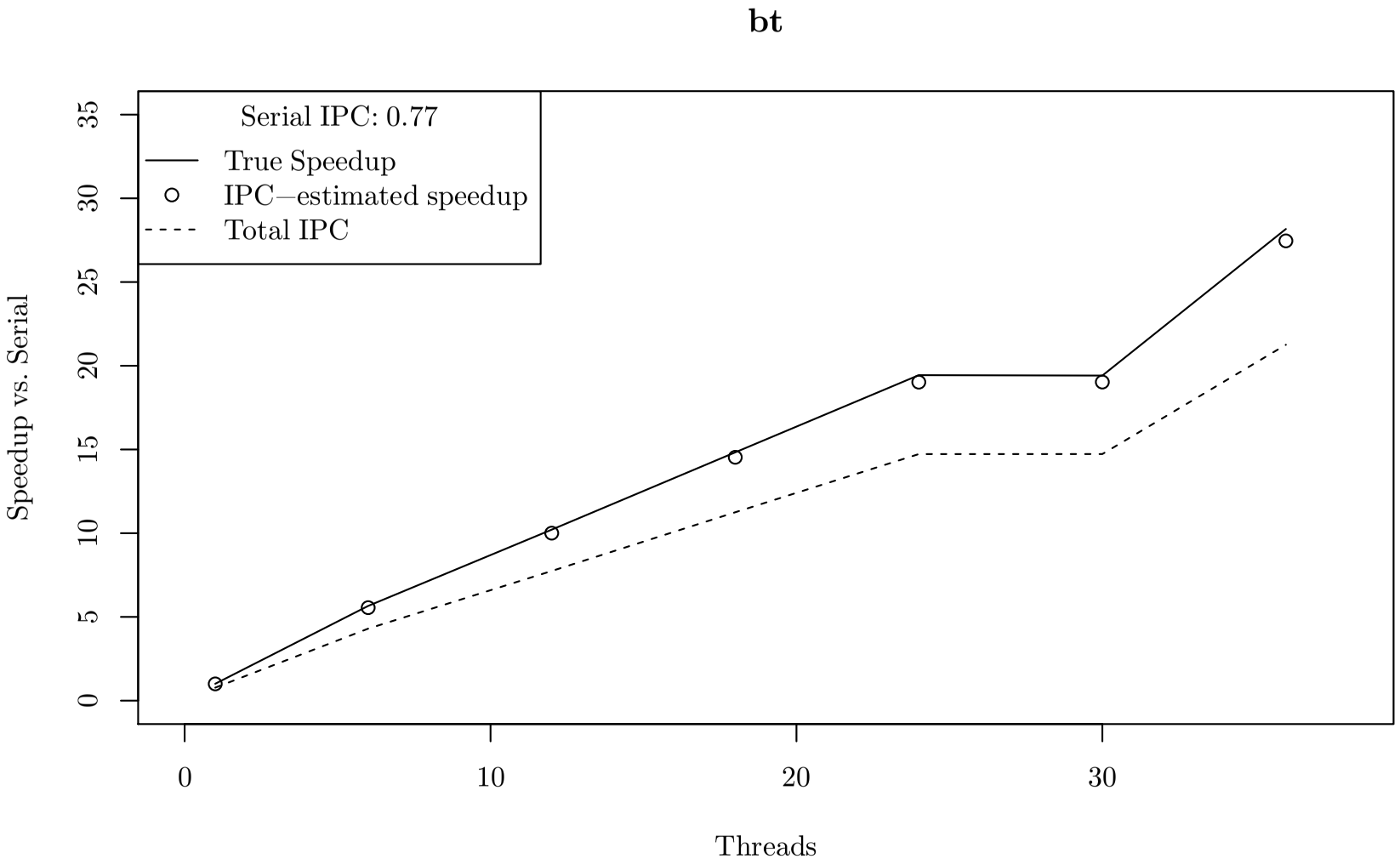


Figure A.6: Comparison of IPC ratio (points) to true speedup (lines) on TileGX for the BT benchmark

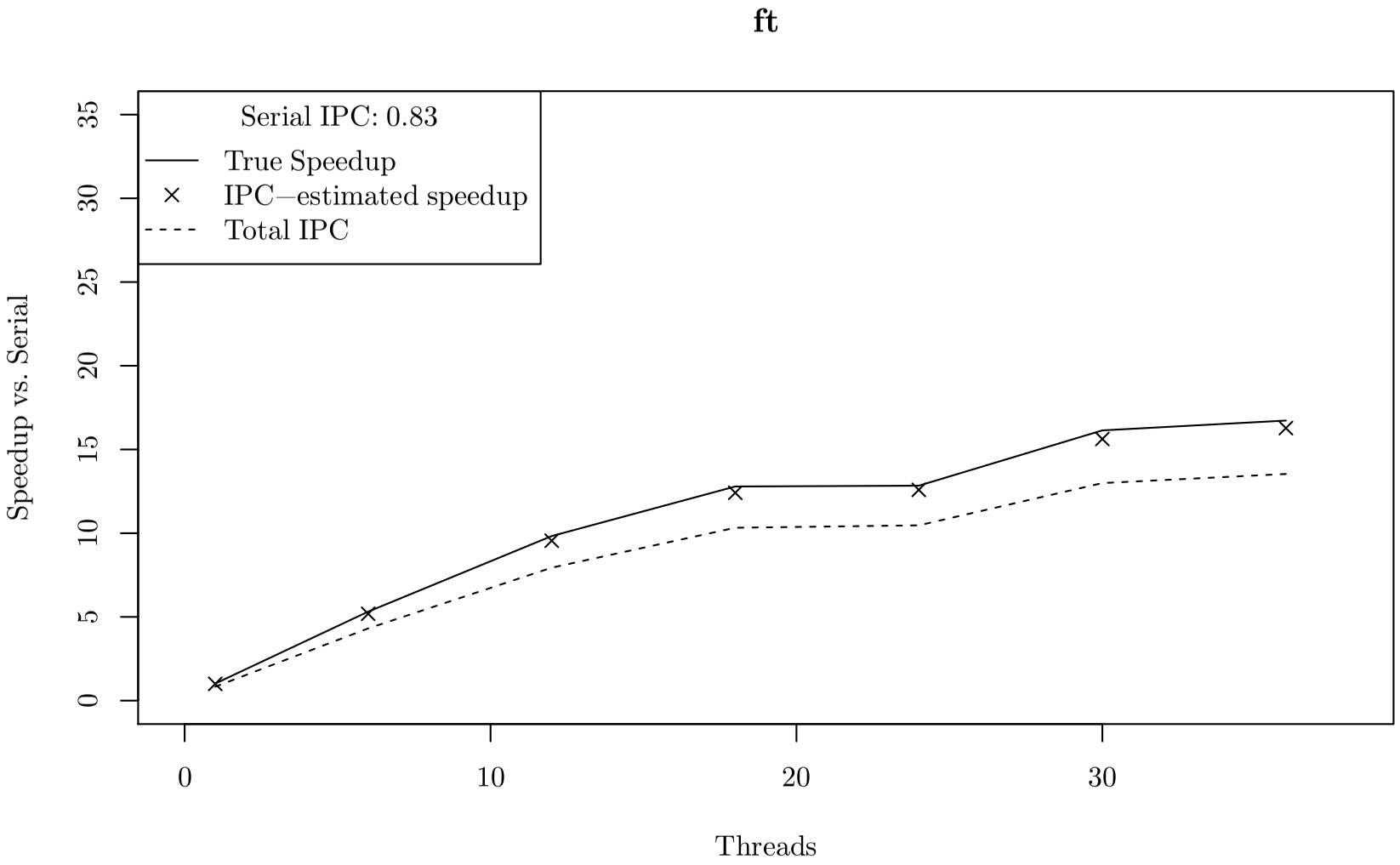


Figure A.7: Comparison of IPC ratio (points) to true speedup (lines) on TileGX for the FT benchmark

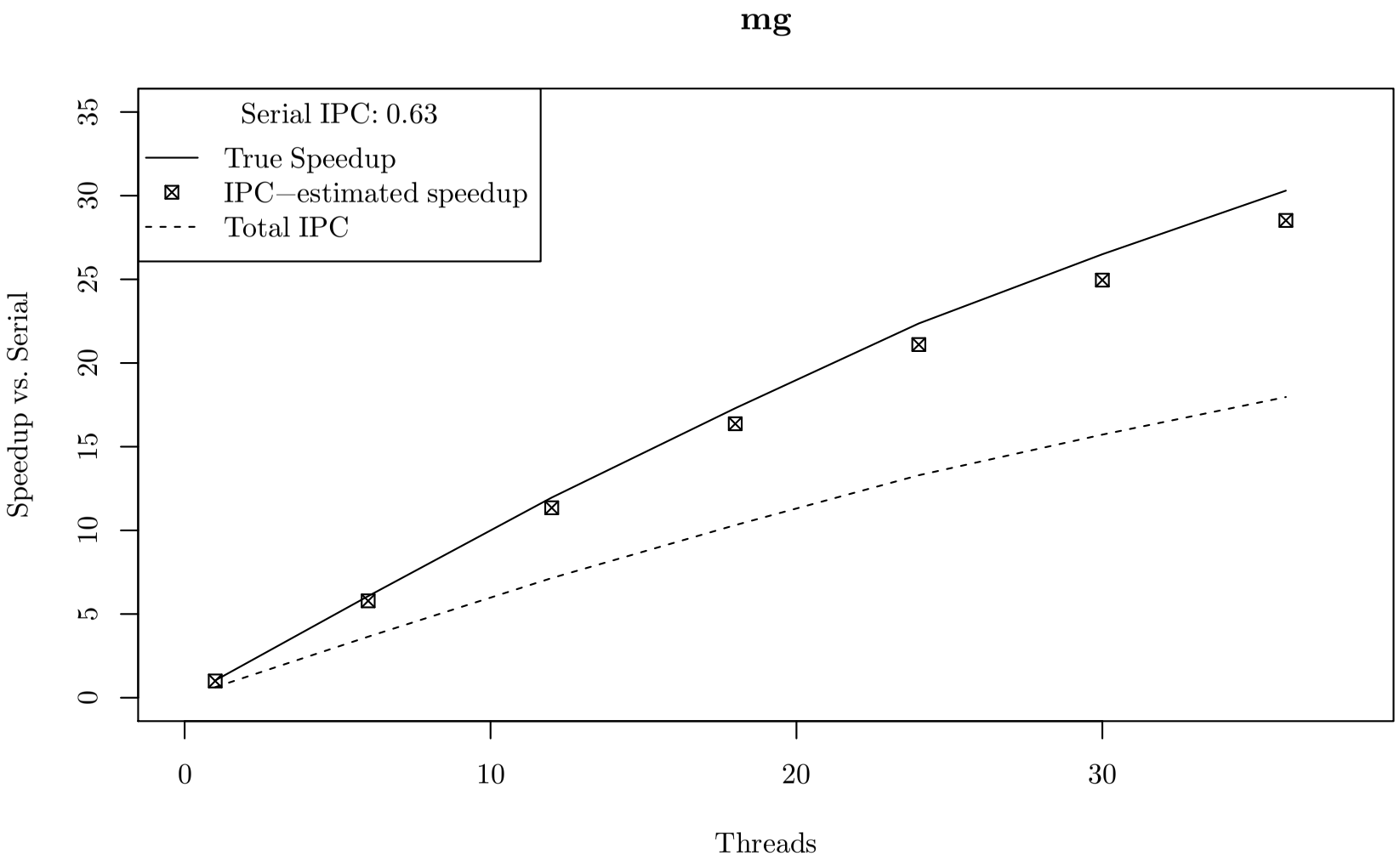


Figure A.8: Comparison of IPC ratio (points) to true speedup (lines) on TileGX for the MG benchmark

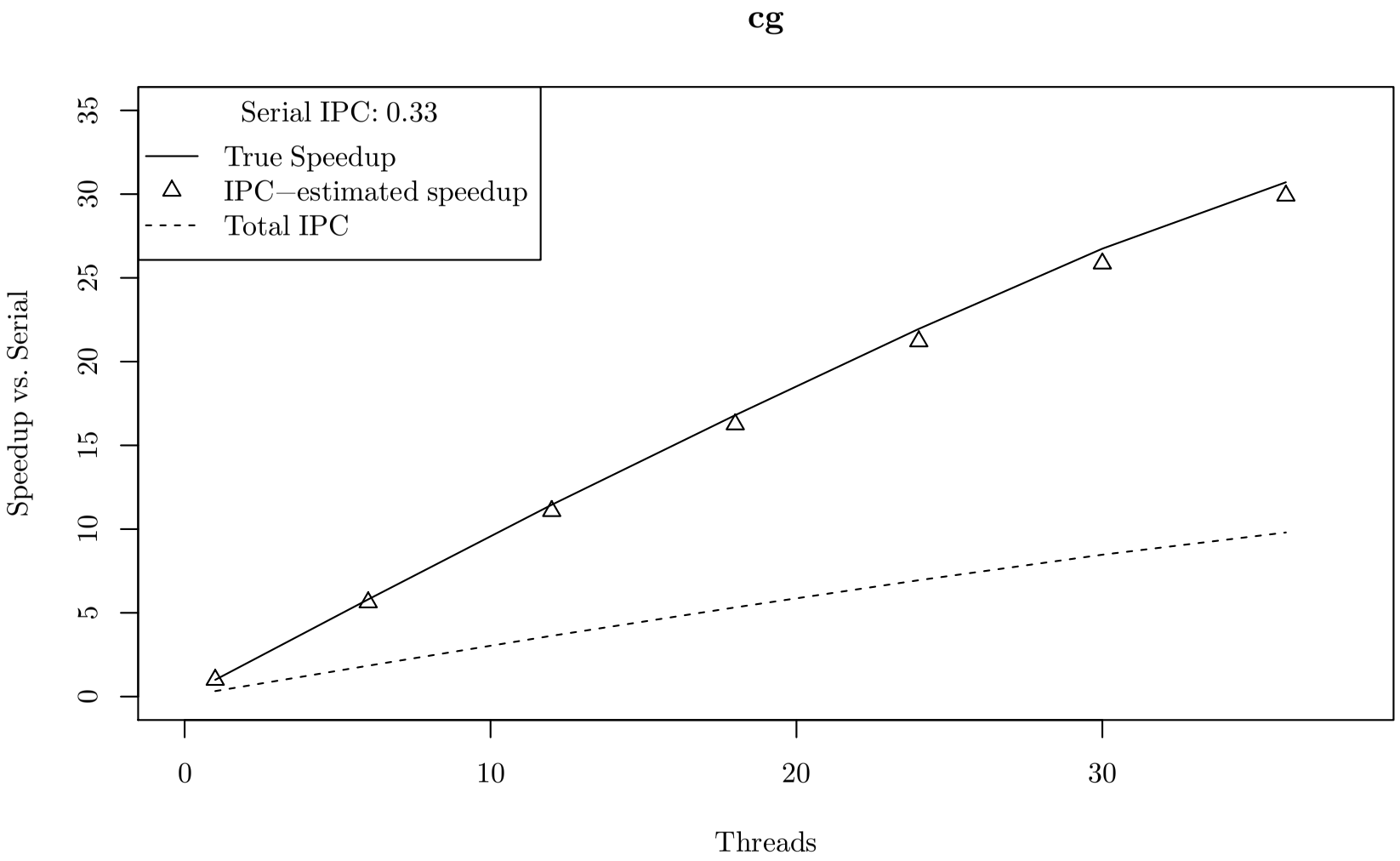


Figure A.9: Comparison of IPC ratio (points) to true speedup (lines) on TileGX for the CG benchmark

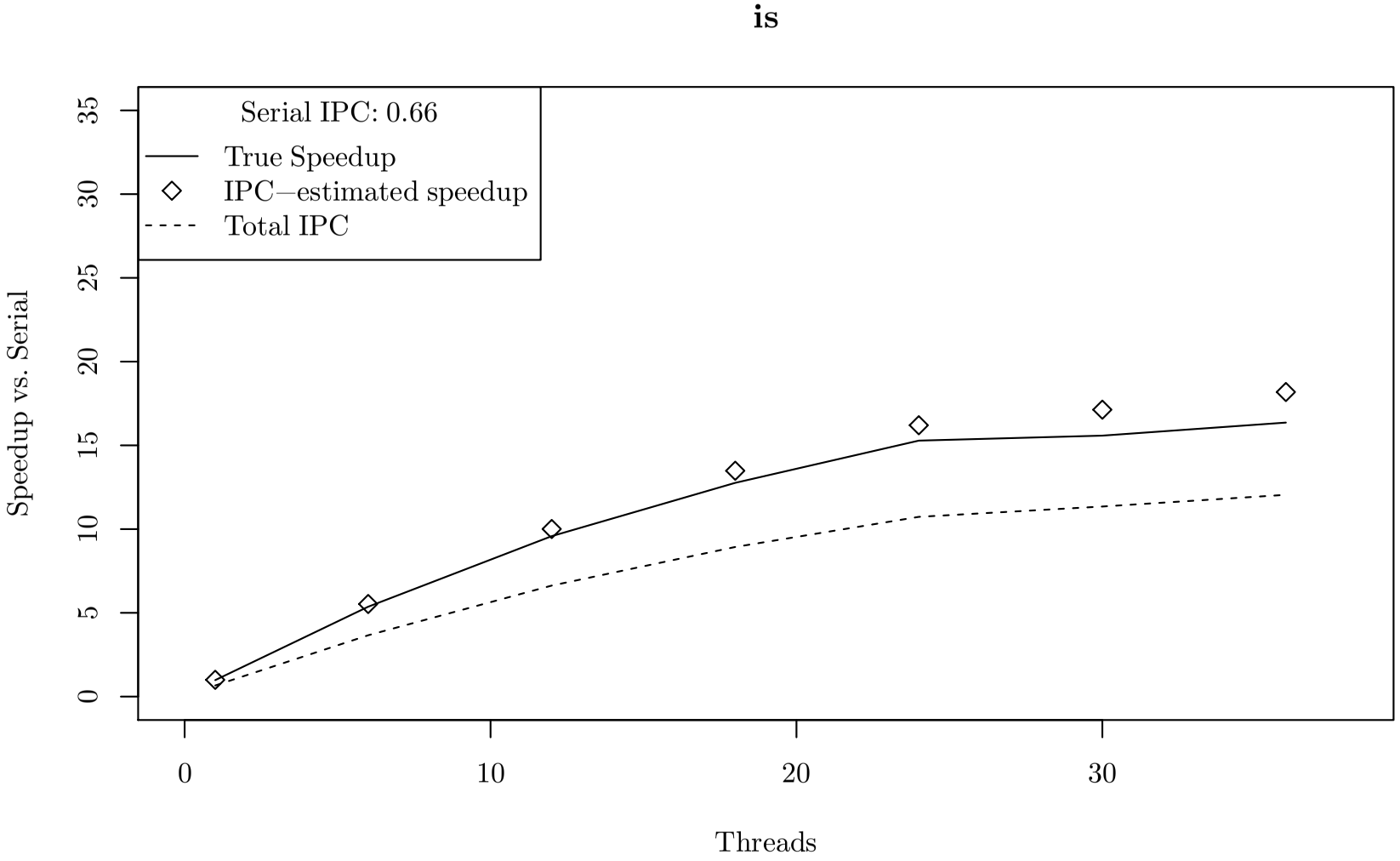


Figure A.10: Comparison of IPC ratio (points) to true speedup (lines) on TileGX for the IS benchmark

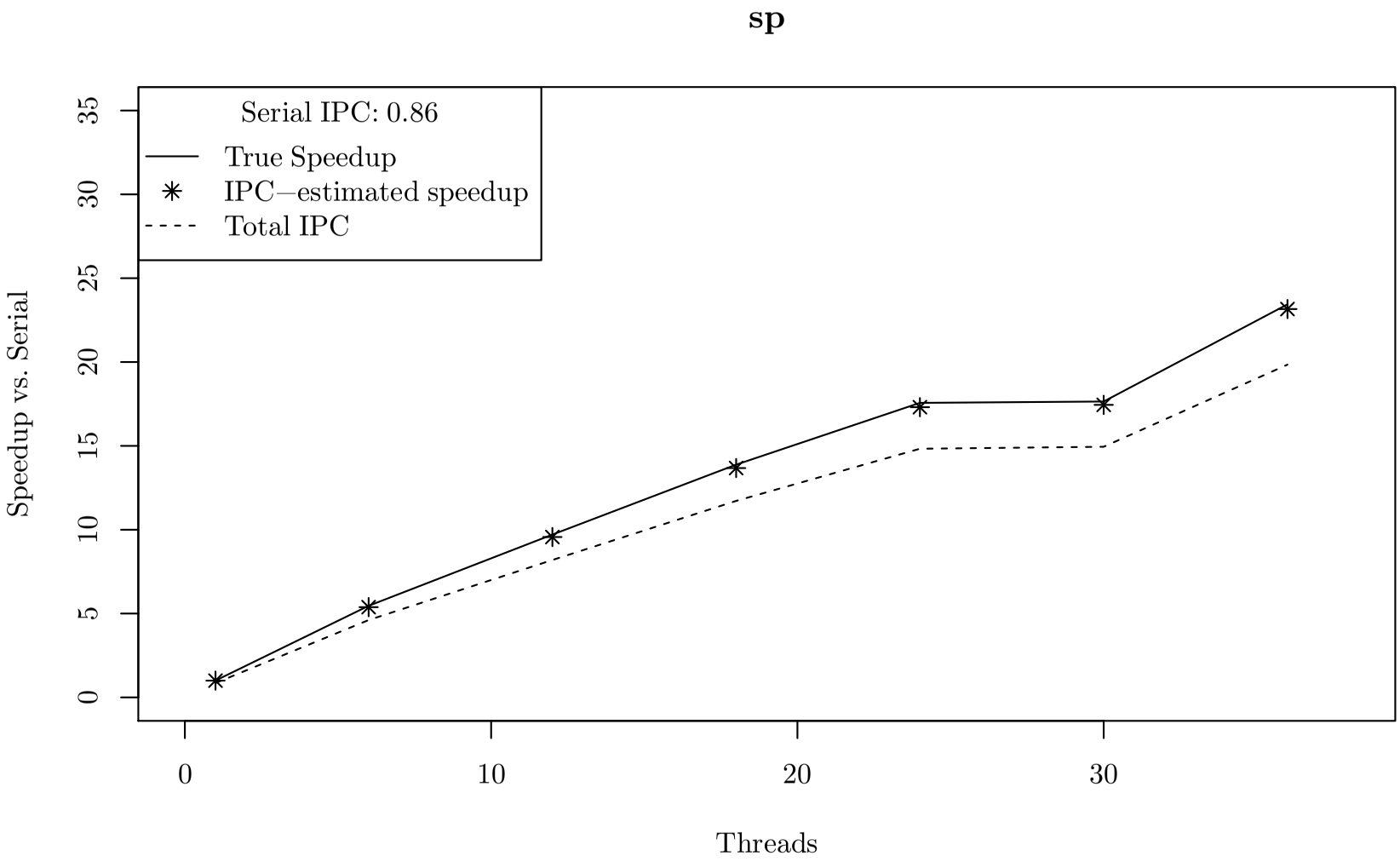


Figure A.11: Comparison of IPC ratio (points) to true speedup (lines) on TileGX for the SP benchmark

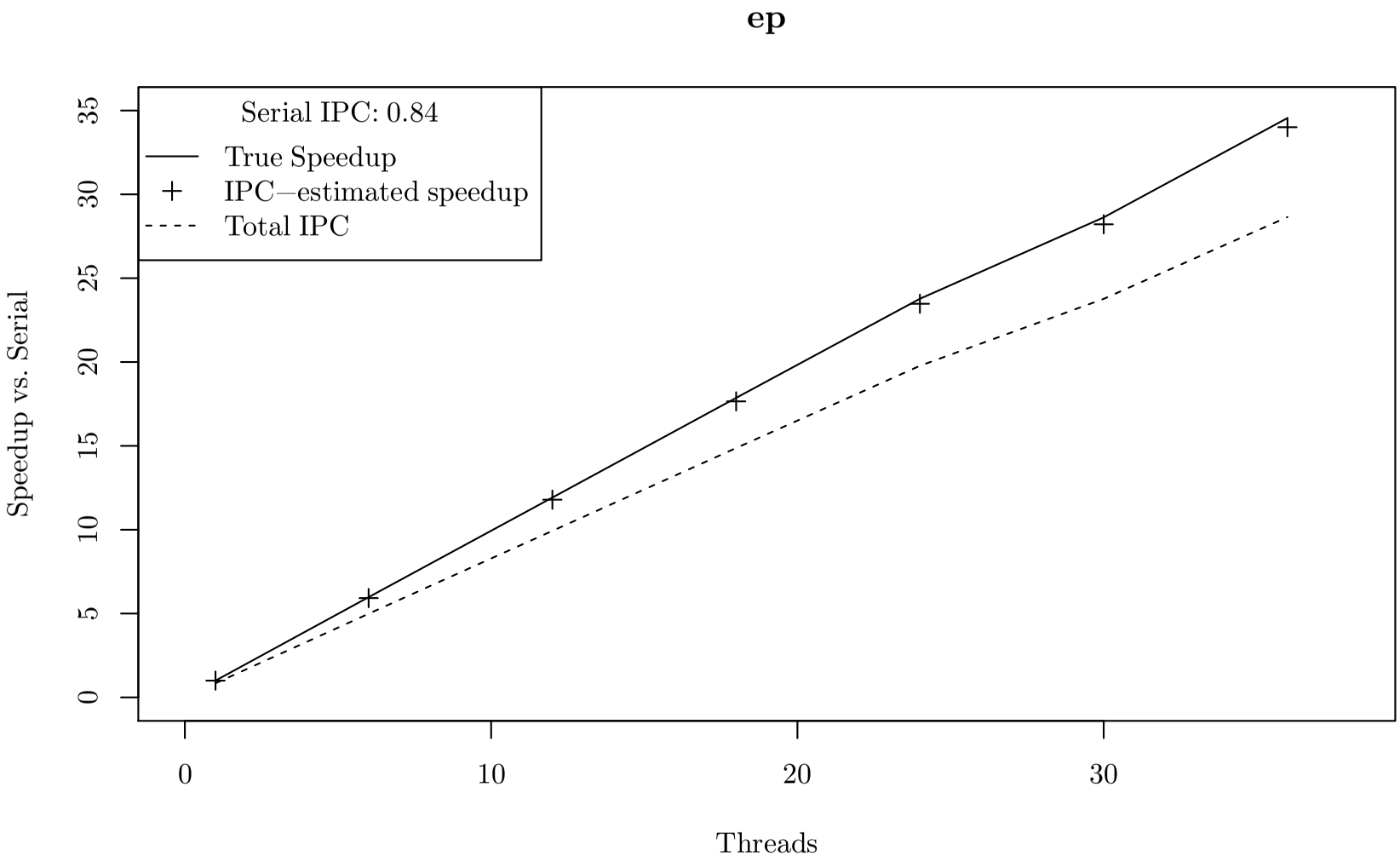


Figure A.12: Comparison of IPC ratio (points) to true speedup (lines) on TileGX for the EP benchmark

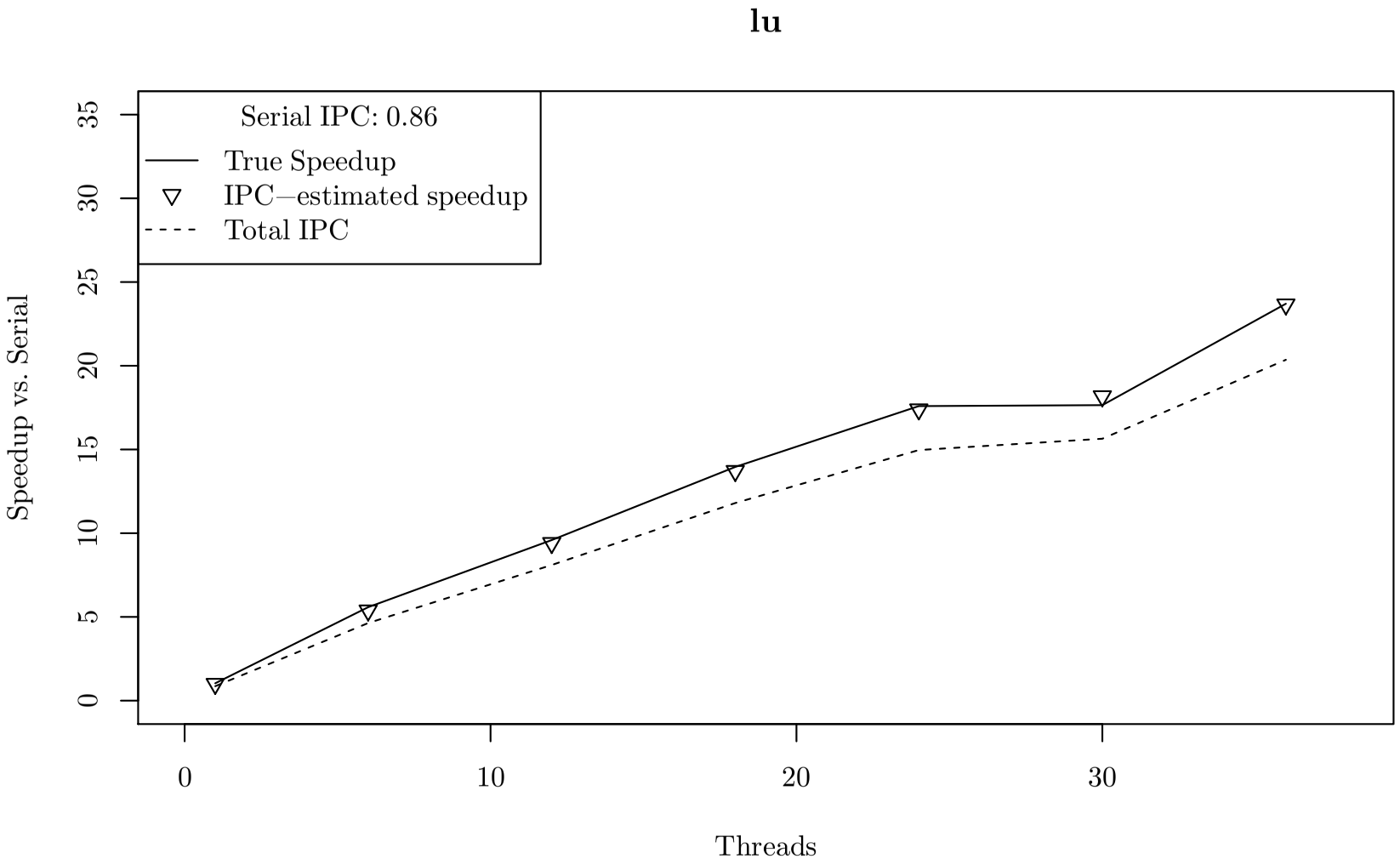


Figure A.13: Comparison of IPC ratio (points) to true speedup (lines) on TileGX for the LU benchmark

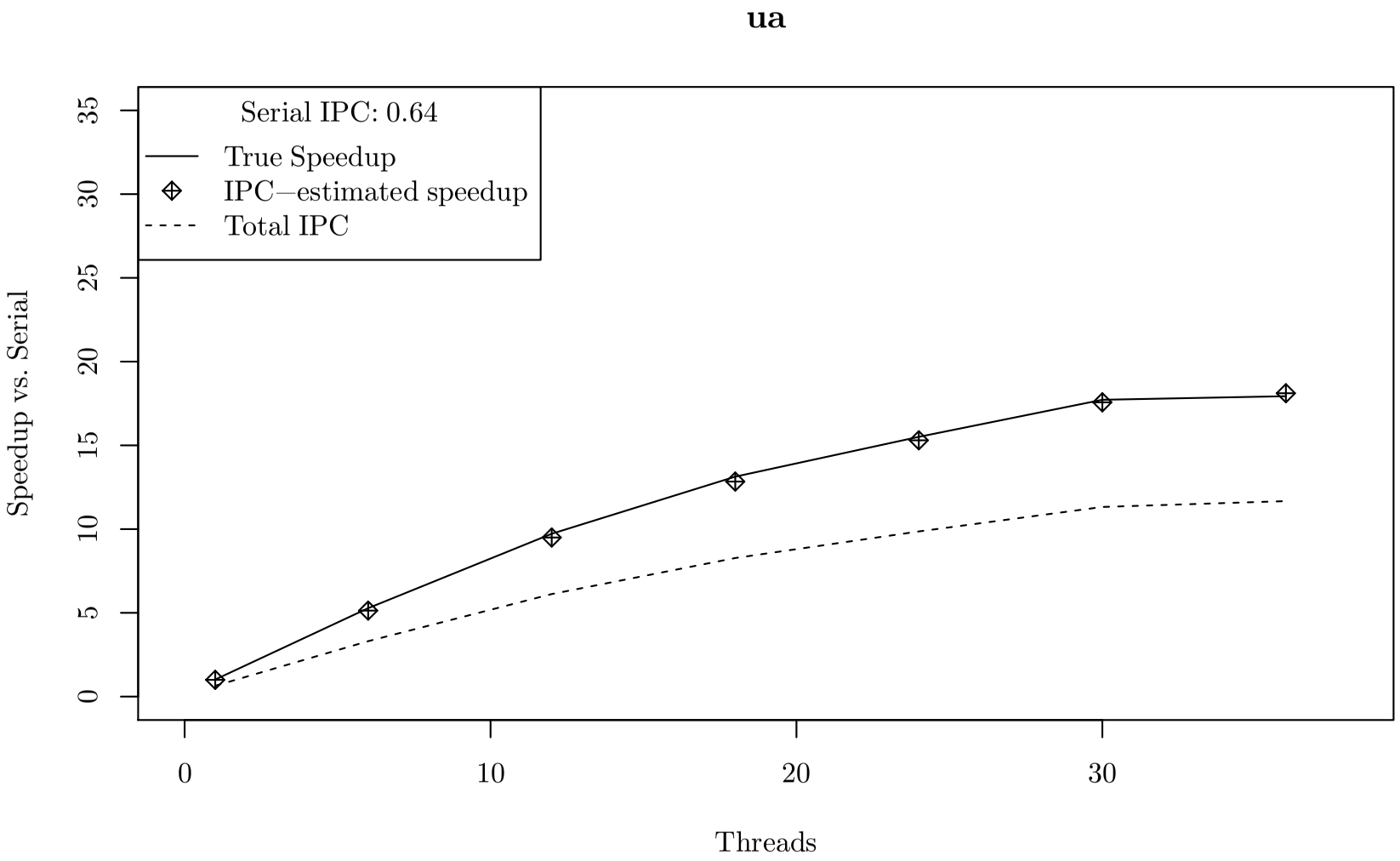


Figure A.14: Comparison of IPC ratio (points) to true speedup (lines) on TileGX for the UA benchmark

Bibliography

[1] Dror G Feitelson and Larry Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Job Scheduling Strategies for Parallel Processing*, pages 1–26. Springer, 1996.

[2] Mary W Hall and Margaret Martonosi. Adaptive parallelism in compiler-parallelized code. *Concurrency: Practice and Experience*, 10(14):1235–1250, December 1998.

[3] I. H Kazi and D. J Lilja. A comprehensive dynamic processor allocation scheme for multiprogrammed multiprocessor systems. In *2000 International Conference on Parallel Processing, 2000. Proceedings*, pages 153–161. IEEE, 2000.

[4] Daniel James McFarland. *Exploiting Malleable Parallelism on Multicore Systems*. University Libraries, Virginia Polytechnic Institute and State University, [Blacksburg, Va, 2011.

[5] Jan H. Schonherr, Jan Richling, and Hans-Ulrich Heiss. Dynamic teams in OpenMP. In *Dynamic Teams in OpenMP*, pages 231–237. IEEE, October 2010.

[6] Jan Hungershöfer, Achim Streit, and Jens-michael Wierum. *Efficient Resource Management for Malleable Applications*. Citeseer, 2001.

[7] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the twelfth ACM symposium on Operating systems principles*, SOSP '89, pages 159–166, New York, NY, USA, 1989. ACM.

[8] Peter B Galvin, Greg Gagne, and Abraham Silberschatz. *Operating system concepts*. John Wiley & Sons, Inc., 2013.

[9] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Russell L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[10] Paul Menage. CGROUPS. `https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`.

[11] Dima Kogan. feedgnuplot: General purpose pipe-oriented plotting tool. `https://github.com/dkogan/feedgnuplot`.

[12] Thomas Williams, Colin Kelley, and many others. Gnuplot: an interactive plotting program. `http://gnuplot.sourceforge.net/`.

[13] Robert D. Blumofe and Dionisios Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. Technical report, The University of Texas at Austin, 1998.

[14] Heidi Pan, Benjamin Hindman, and Krste Asanović. Composing parallel software efficiently with lithe. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 376–387, New York, NY, USA, 2010. ACM.

[15] Jeremy Buisson, Omer Ozan Sonmez, Hashim H. Mohamed, Wouter Lammers, and Dick H. J. Epema. Scheduling malleable applications in multicluster systems. In *CLUSTER*, pages 372–381. IEEE Computer Society, 2007.

[16] Sathish S. Vadhiyar and Jack Dongarra. SRS: A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems. *Parallel Processing Letters*, 13(2):291–312, 2003.

[17] Liang Chen, Qian Zhu, and Gagan Agrawal. Supporting Dynamic Migration in Tightly Coupled Grid Applications. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[18] Rohit Fernandes, Keshav Pingali, and Paul Stodghill. Mobile MPI Programs in Computational Grids. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 22–31, New York, NY, USA, 2006. ACM.

[19] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, and Laxmikant Kale. Parallel Programming with Migratable Objects: Charm++ in Practice. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 647–658, Piscataway, NJ, USA, 2014. IEEE Press.

[20] Jeremy Buisson, Francoise Andre, and Jean-Louis Pazat. Supporting Adaptable Applications in Grid Resource Management Systems. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, GRID '07, pages 58–65, Washington, DC, USA, 2007. IEEE Computer Society.

[21] Travis J. Desell, Kaoutar El Maghraoui, and Carlos A. Varela. Malleable applications for scalable high performance computing. *Cluster Computing*, 10(3):323–337, 2007.

[22] Walfredo Cirne and Francine Berman. A Model for Moldable Supercomputer Jobs. In *IPDPS*, page 59. IEEE Computer Society, 2001.

[23] Cristian Klein and Christian Pérez. An RMS Architecture for Efficiently Supporting Complex-Moldable Applications. In Parimala Thulasiraman, Laurence Tianruo Yang, Qiwen Pan, Xingang Liu, Yaw-Chung Chen, Yo-Ping Huang, Lin-Huang Chang, Che-Lun Hung, Che-Rung Lee, Justin Y. Shi, and Ying Zhang, editors, *HPCC*, pages 211–220. IEEE, 2011.

[24] Kaoutar El Maghraoui, Travis J. Desell, Boleslaw K. Szymanski, and Carlos A. Varela. Dynamic Malleability in Iterative MPI Applications. In *CCGRID*, pages 591–598. IEEE Computer Society, 2007.

[25] Gladys Utrera, Julita Corbalán, and Jesús Labarta. Implementing Malleability on MPI Jobs. In *IEEE PACT*, pages 215–224. IEEE Computer Society, 2004.

[26] Matthias Hovestadt, Odej Kao, Axel Keller, and Achim Streit. Scheduling in HPC Resource Management Systems: Queuing vs. Planning. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *JSSPP*, volume 2862 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2003.

[27] Andrew D. Ferguson, Peter Bodík, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: guaranteed job latency in data parallel clusters. In Pascal Felber, Frank Bellosa, and Herbert Bos, editors, *EuroSys*, pages 99–112. ACM, 2012.

[28] Gonzalo Pedro Rodrigo Álvarez, Per-Olov Östberg, Erik Elmroth, and Lavanya Ramakrishnan. A2L2: An Application Aware Flexible HPC Scheduling Model for Low-Latency Allocation. In *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*, VTDC '15, pages 11–19, New York, NY, USA, 2015. ACM.

[29] Hesham El-Rewini and Ted G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of parallel and Distributed Computing*, 9(2):138–153, 1990.

[30] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, 2002.

[31] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Ladislau Bölöni, Muthucumaru Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, Bin Yao, Debra A. Hensgen, and Richard F. Freund. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *J. Parallel Distrib. Comput.*, 61(6):810–837, 2001.

[32] Bobby Dalton Young, Sudeep Pasricha, Anthony A. Maciejewski, Howard Jay Siegel, and James T. Smith. Heterogeneous Makespan and Energy-constrained DAG Scheduling. In *Proceedings of the 2013 Workshop on Energy Efficient High Performance Parallel and Distributed Computing*, EEHPDC '13, pages 3–12, New York, NY, USA, 2013. ACM.

[33] Saeid Abrishami, Mahmoud Naghibzadeh, and Dick H. J. Epema. Deadline-constrained workflow scheduling algorithms for Infrastructure as a Service Clouds. *Future Generation Comp. Syst.*, 29(1):158–169, 2013.

[34] Young Choon Lee and Albert Y. Zomaya. Energy Conscious Scheduling for Distributed Computing Systems under Different Operating Conditions. *IEEE Trans. Parallel Distrib. Syst.*, 22(8):1374–1381, 2011.

[35] Rajkumar Buyya. Economic-based Distributed Resource Management and Scheduling for Grid Computing. *CoRR*, cs.DC/0204048, 2002.

[36] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, GrÃ©gory MouniÃ©, Pierre Neyron, and Olivier Richard. A batch scheduler with high level components. *Cluster computing and Grid 2005 (CCGrid05), Royaume-Uni (2005)*, June 2005.

[37] Prakhar Gupta, Tarun Atrey, Manjari Garg, Verdi March, and Simon Chong Wee See. Batch scheduler for personal multi-core systems. In *Distributed Computing and Applications to Business Engineering and Science (DCABES), 2010 Ninth International Symposium on*, pages 584–587. IEEE, 2010.

[38] L. V Kale, S. Kumar, and J. DeSouza. A Malleable-Job system for timeshared parallel machines. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2002*, pages 230– 230. IEEE, May 2002.

[39] R. Sudarsan and C.J. Ribbens. ReSHAPE: a framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment. In *Parallel Processing, 2007. ICPP 2007. International Conference on*, page 44, September 2007.

[40] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 119–129, New York, NY, USA, 1998. ACM.

[41] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.

[42] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 277–286, New York, NY, USA, 2008. ACM.

[43] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. Thread reinforcer: Dynamically determining number of threads via OS level monitoring. In *IISWC*, pages 116–125. IEEE Computer Society, 2011.

[44] Janghaeng Lee, Haicheng Wu, Madhumitha Ravichandran, and Nathan Clark. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. In André Seznec, Uri C. Weiser, and Ronny Ronen, editors, *ISCA*, pages 270–279. ACM, 2010.

[45] Changhee Jung, Daeseob Lim, Jaejin Lee, and Sangyong Han. Adaptive execution techniques for SMT multiprocessor architectures. In Keshav Pingali, Katherine A. Yelick, and Andrew S. Grimshaw, editors, *PPOPP*, pages 236–246. ACM, 2005.

[46] Yang Ding, Mahmut T. Kandemir, Padma Raghavan, and Mary Jane Irwin. A helper thread based EDP reduction scheme for adapting application execution in CMPs. In *IPDPS*, pages 1–14. IEEE, 2008.

[47] Jaejin Lee and H. D. K. Moonesinghe. Adaptively Increasing Performance and Scalability of Automatically Parallelized Programs. In William Pugh and Chau-Wen Tseng, editors, *LCPC*, volume 2481 of *Lecture Notes in Computer Science*, pages 203–217. Springer, 2002.

[48] M. Voss and R. Eigenmann. Reducing parallel overheads through dynamic serialization. In *Parallel and Distributed Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings*, pages 88–92. IEEE, April 1999.

[49] Matthew Curtis-Maury, James Dzierwa, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In Gregory K. Egan and Yoichi Muraoka, editors, *ICS*, pages 157–166. ACM, 2006.

[50] Matthew Curtis-Maury, Karan Singh, Sally A. McKee, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. Identifying energy-efficient concurrency levels using machine learning. In *CLUSTER*, pages 488–495. IEEE Computer Society, 2007.

[51] Bradley J. Barnes, Barry Rountree, David K. Lowenthal, Jaxk Reeves, Bronis R. de Supinski, and Martin Schulz. A regression-based approach to scalability prediction. In Pin Zhou, editor, *ICS*, pages 368–377. ACM, 2008.

[52] Laxmikant V Kale, Josh Yelon, and Timothy Knauff. Threads for interoperable parallel programming. In *Languages and Compilers for Parallel Computing*, pages 534–552. Springer, 1997.

[53] Peng Li, Simon Marlow, Simon L. Peyton Jones, and Andrew P. Tolmach. Lightweight concurrency primitives for GHC. In Gabriele Keller, editor, *Haskell*, pages 107–118. ACM, 2007.

[54] Matthew Fluet, Mike Rainey, and John H. Reppy. A scheduling framework for general-purpose parallel languages. In James Hook and Peter Thiemann, editors, *ICFP*, pages 241–252. ACM, 2008.

[55] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-Class User-Level Threads. *SIGOPS Oper. Syst. Rev.*, 25(5):110–121, 1991.

[56] T.E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[57] Alaa R Alameldeen and David A Wood. Ipc considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4):8–17, 2006.

[58] ICL, University of Tennessee and contributors. Performance Application Programming Interface (PAPI), 2012.

[59] A. B. Downey. *A model for speedup of parallel programs*. University of California, Berkeley, Computer Science Division, 1997.

[60] Su-Hui Chiang, Rajesh K. Mansharamani, and Mary K. Vernon. Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies. *SIGMETRICS Perform. Eval. Rev.*, 22(1):33–44, May 1994.

[61] GCC 4.6.4 GNU OpenMP Manual. `https://gcc.gnu.org/onlinedocs/gcc-4.6.4/libgomp/`. Accessed: 2015-09-18.

[62] ARB OpenMP. OpenMP application program interface version 4.0, 2013.

[63] iMatix Corporation and contributors. ZeroMQ, 2012.

[64] Troy D Hanson. uthash User Guide. *URL: https://troydhanson.github.io/uthash/userguide.html*, 2014.

[65] TILE-Gx36 Processor Product Brief. `http://www.tilera.com/files/drim_TILE-Gx8036_PB033-03_WEB_7682.pdf`. Accessed: 2015-09-18.

[66] Intel® Xeon® Processor E5-2690 (20M Cache, 2.90 GHz, 8.00 GT/s Intel® QPI). `http://ark.intel.com/products/64596/Intel-Xeon-Processor-E5-2690-20M-Cache-2_90-GHz-8_00-GTs-Intel-QPI`. Accessed: 2015-09-18.

[67] Intel® Xeon Phi™ Coprocessor 5110P (8GB, 1.053 GHz, 60 core). `http://ark.intel.com/products/71992/Intel-Xeon-Phi-Coprocessor-5110P-8GB-1_053-GHz-60-core`. Accessed: 2015-09-18.

[68] Manish Shah, J Barren, Jeff Brooks, Robert Golla, Gregory Grohoski, Nils Gura, Rick Hetherington, Paul Jordan, Mark Luttrell, Christopher Olson, et al. UltraSPARC T2: A highly-treaded, power-efficient, SPARC SOC. In *Solid-State Circuits Conference, 2007. ASSCC'07. IEEE Asian*, pages 22–25. IEEE, 2007.

[69] Opteron 6212. `http://products.amd.com/en-us/search/CPU/AMD-Opteron%E2%84%A2/AMD-Opteron%E2%84%A2-6200-Series-Processor/6212/35`. Accessed: 2015-09-18.

[70] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 180–186. IEEE, 2010.