Washington University in St. Louis
Washington University Open Scholarship

McKelvey School of Engineering Theses & Dissertations | McKelvey School of Engineering

Winter 12-15-2017

Locality-Aware Concurrency Platforms

Jordyn Chrystopher Raymond Maglalang
Washington University in St. Louis

Follow this and additional works at: https://openscholarship.wustl.edu/eng_etds

Part of the Computer Sciences Commons

Recommended Citation

Maglalang, Jordyn Chrystopher Raymond, "Locality-Aware Concurrency Platforms" (2017). *McKelvey School of Engineering Theses & Dissertations*. 290.
https://openscholarship.wustl.edu/eng_etds/290

This Dissertation is brought to you for free and open access by the McKelvey School of Engineering at Washington University Open Scholarship. It has been accepted for inclusion in McKelvey School of Engineering Theses & Dissertations by an authorized administrator of Washington University Open Scholarship. For more information, please contact digital@wumail.wustl.edu.

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science
Department of Computer Science and Engineering

Dissertation Examination Committee:
Kunal Agrawal, Chair
Roger Chamberlain
Chris Gill
Sriram Krishnamoorthy
I-Ting Angelina Lee

Locality-Aware Concurrency Platforms

by

Jordyn Chrystopher Raymond Maglalang

A dissertation presented to
The Graduate School
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

December 2017
Saint Louis, Missouri

© 2017, Jordyn Chrystopher Raymond Maglalang

Table of Contents

List of Figures . . . v

List of Tables . . . viii

Acknowledgments . . . ix

Abstract . . . xi

Chapter 1: Introduction . . . 1
1.1 Streaming Applications . . . 3
1.1.1 The Streaming Model . . . 4
1.1.2 Assumptions and Definitions . . . 5
1.2 Dynamic Multithreading . . . 6

Chapter 2: Cache-Concious Scheduling of Streaming Pipelines . . . 8
2.1 Model . . . 9
2.2 Theory . . . 10
2.2.1 Lower Bound on Total Misses . . . 10
2.2.2 Lower Bound on Time . . . 12
2.2.3 The *seg_cache* Algorithm . . . 15
2.2.4 Bounding *seg_cache*'s Performance . . . 17
2.3 Conclusions and Future Work . . . 19

Chapter 3: Executing Streaming Pipelines . . . 20
3.1 Scheduling Policies . . . 20
3.2 Pipeline Execution . . . 22
3.3 Experiments . . . 23
3.3.1 Pipeline Generation . . . 23
3.3.2 Experimental Settings . . . 24
3.3.3 Disabled Computation . . . 25
3.3.4 Randomly Generated Computation Times . . . 26
3.3.5 Correlated Computation Time . . . 28
3.3.6 Segmentation Based on Computation Time and Cache . . . 29
3.3.7 Fixed State and Computation . . . 32

3.4 Related Work 33
3.5 Conclusions and Future Work 34

Chapter 4: AMPipe: Automatically Mapping Streaming Pipelines 35
4.1 Intro 35
4.1.1 Contributions 36
4.2 Design and Implementation 38
4.2.1 AMPIPE front-end 39
4.2.2 Mapping and Code Generation 42
4.2.3 Runtime 46
4.3 Experiments 49
4.4 Related Work 53
4.5 Conclusions 55

Chapter 5: Locality-Aware Dynamic Task-Graph Scheduling 64
5.1 Intro 64
5.2 Background 66
5.3 Theory 70
5.4 Design 71
5.5 Experiments 78
5.5.1 Overall performance 81
5.5.2 Locality impact of NABBITC's scheduling strategy 82
5.5.3 Overheads due to colored steals 83
5.5.4 Importance of good coloring 84
5.6 Related Work 85
5.7 Conclusions and Future Work 87

Chapter 6: Locality-Friendly Parallel For Loops 95
6.1 Introduction 95
6.2 Related Work 97
6.3 Background 98
6.4 Design 99
6.4.1 Assigning Iterations to Workers 100
6.4.2 Elevating CILK_PARFOR beyond syntactic sugar 101
6.4.3 Stealing iterations towards load balance 101
6.5 Implementation 102
6.5.1 Programming Interface 102
6.5.2 CILK_PARFOR loop execution 103
6.6 Experiments 111
6.6.1 Experimental Setup 111
6.6.2 Time-Bound Applications 112
6.6.3 Data-Bound Applications 114
6.6.4 Investigating Our Overhead 115

6.7 Conclusions and Future Work 116

References 118

List of Figures

Figure 1.1: An example pipeline with annotated state size, input and output rates and gains for both modules and edges. 4

Figure 3.1: Normalized results for *seg_rand* and *rand_assign* against *seg_cache* with computation simulation disabled. We see that *seg_cache* works better with zipf gain than with uniform gain. 25

Figure 3.2: Normalized results for *seg_runtime*, *seg_random*, *bin_fullest* and *bin_-emptiest* against *seg_cache* with computation time enabled. We see that *seg_cache* performs the best when both the computation time and the gains have a zipf's distribution. This is due to the fact that (1) zipf's distribution on computation time reduces the total computation time of the pipeline, making it more likely that the runtime is dominated by cache misses and (2) zipf's distribution on gains allows *seg_cache* more leeway to pick better cross edges. 27

Figure 3.3: *seg_runtime*, *seg_random*, *bin_fullest* and *bin_emptiest* against *seg_cache* for correlated state size and computation time. 29

Figure 3.4: Segmentation based on both computation time and cache. We see that this form of segmentation does a better job of load-balancing computations when computation-based strategies dominate, but also gets most of the advantage of cache-based segmentation when *seg_cache* dominates. . . 30

Figure 3.5: Normalized results *seg_runtime*, *seg_random*, *bin_fullest* and *bin_emptiest* against *seg_cache* when state size and computation times have been fixed and gains are selected with a zipf distribution. 31

Figure 4.1: A pipeline example with various mappings. 39

Figure 4.2: Architecture of the AMPIPE platform. 40

Figure 4.3: A full "Hello, World" pipeline programming demonstrating kernel,pipeline and application definitions including the newly required calls to pass profiling and system information to the front-end. 56

Figure 4.4: AMPIPE's abstract Mapper class which provides the interface for defining new Mappers which at a minimum requires users to implement the create_segments and assign_segments_to_cores methods. 57

Figure 4.5: The SEGBOTH segmentation algorithm. schedule_maxload is a helper function which determines if a segmentation can be found within a given load and if so, stores it. 58

Figure 4.6: Kernel scheduling function within a segment. Each firing results in the first kernel of the segment executing once before propataing it's output as far as possible through the segment. 59

Figure 4.7: AMPIPE's single producer single consumer thread safe ring buffer used for cross edges between non-replicated modules. 60

Figure 4.8: AMPIPE's Split and Join edges, which are responsible for trafficking data between replicated and non-replicated kernels. Only one producer will be connected to a split edge, while only one consumer will be connected to a join edge, making the read and write methods unnecessary in those two classes respectively. 61

Figure 4.9: The Interchange and ITSPQ classes responsible for trafficking data between two replicated kernels 62

Figure 5.1: NABBIT scheduling examples presented in two stages. A thread surrounding a node denotes that thread is processing that node. A solid line from a to b denotes that b is a predecessor of a while a dashed line from b to a denotes that a is in b's successor list. 66

Figure 5.2: NABBITC abstract class interface 72

Figure 5.4: Key routines to spawn predecessors and successors in NABBITC. 90

Figure 5.5: Extensions to the Cilk Plus RTS API to inform the runtime of the worker's preferred color and the colors available in a continuation. 90

Figure 5.6: Speedup for all benchmarks. x-axis: number of threads (processor cores); y-axis: speedup over serial OPENMPSTATIC. 92

Figure 5.7: Percentage of accesses that are to data in remote NUMA domains. We show percentages for 20–80 cores (1–10 cores fit in one NUMA domain and do not incur remote accesses). x-axis: core count; y-axis: Percentage of accesses that are remote. 92

Figure 5.8: Average number of successful steals for (a) NABBITC and (b) NABBIT. x-axis: number of cores; y-axis: Average number of successful steals . . 93

Figure 5.9: Average idle time per processor core (across all processor cores and runs) due to forcing the first colored steal for the heat benchmark. Error bars show standard deviation across five runs among all processor cores. We observed this time was the same for all benchmarks. 93

Figure 6.1: CILK_PARFOR user interface 102

Figure 6.2: The expansion of the CILK_PARFOR_BEGIN and CILK_PARFOR_END macros 104

Figure 6.3: The main CILK_PARFOR execution body 105

Figure 6.4: Iteration steal example 109

Figure 6.5: Smith-Waterman results 113

Figure 6.6: Conway's Game of Life results 114

List of Tables

Table 4.1: Runtimes (with speedup over serial in parenthesis) for DES, LZ77, Ferret, and DNN. sr is SEGRUNTIME without replication, srr is SEGRUNTIME with replication, sb is SEGBOTH without replication and sbr is SEGBOTH with replication. For DES and DNN, mappers with and without replication find the same mapping. 50

Table 4.2: L2 Cache miss counts for DNN with for SEGRUNTIME and SEGBOTH . 53

Table 5.1: Benchmark configurations and serial OPENMPSTATIC execution time. The PageRank benchmarks use the same code with three different web crawl datasets [18, 19, 46]. 91

Table 5.2: Speedup of NABBITC over NABBIT when all tasks are assigned bad colors resulting in preferential execution of non-local tasks. S.D. denotes standard deviation. 93

Table 5.3: Speedup of NABBITC over NABBIT when all tasks are assigned invalid colors resulting in failure of all colored steal attempts. S.D. denotes standard deviation. 94

Acknowledgments

This work would not have been possible without the help and support of all of my colleagues, friends and family. I could fill the entirety of this document with thanks to everyone who's played a role in getting me here, indeed there could never be enough space. Although I only mention a few of you, I am forever indebted to you all.

Thank you to my advisor, Kunal Agrawal, for taking me under your wing and providing guidance over the years. To Sriram Krishnamoorthy, for your supervision during my time at PNNL and your help towards securing my first first-author paper. Thank you to the rest of my committee members: Roger Chamberlain, Chris Gill and Angelina Lee. Your encouragement and help has gotten me through each of the milestones leading to this.

I want to thank my fellow lab members in the Parallel Computing Technology Group for all that I've learned from you. Thank you Rob for all the helpful conversations that got me unstuck and reminded me I wasn't always alone in my work. To Adam, thanks for aways popping in with questions or curiosity. You always kept me on my feet or pulled me away to the Kings when what I really needed was a break.

To Michelle, who was a constant voice of support throughout every inch of this roller-coaster effort, I could never thank you enough. We might not have become the the best dart players in the world like we planned, but we helped each other keep our heads above water over these years. Thank you for never letting me give up.

Thank to all my friends all over the place, who've watched me struggle and succeed during this whole ordeal. I relied on you all a lot more than I might have mentioned. This accomplishment is ours together. Thank you Carl, for helping keep my head straight with everything thats happened over the years. Josh, thanks for always having my back and giving me refuge when I needed to escape, you guys remind me that I'm not in this on my own. To Amy, who could see me when so many others couldn't, thank you.

To Janine, who's never let me drift too far out of arms length, thank you. We chose to always be there for each other, despite the challenge of the physical distance. This accomplishment is as much for you, and for us, as it is for me.

To my family, who've all been with me for every step, thank you. You believed in me well before I did, I couldn't have gotten here without you.

This research was supported by National Science Foundation grants CCF-1527692, CCF-143906, and CCF-1150036, and by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under award number 63823.

Jordyn Chrystopher Raymond Maglalang

Washington University in Saint Louis

December 2017

ABSTRACT OF THE DISSERTATION

Locality-Aware Concurrency Platforms
by
Jordyn Chrystopher Raymond Maglalang
Doctor of Philosophy in Computer Science
Washington University in St. Louis, 2017
Professor Kunal Agrawal, Chair

Modern computing systems from all domains are becoming increasingly more parallel. Manufacturers are taking advantage of the increasing number of available transistors by packaging more and more computing resources together on a single chip or within a single system. These platforms generally contain many levels of private and shared caches in addition to physically distributed main memory. Therefore, some memory is more expensive to access than other and high-performance software must consider memory locality as one of the first level considerations.

Memory locality is often difficult for application developers to consider directly, however, since many of these NUMA affects are invisible to the application programmer and only show up in low performance. Moreover, on parallel platforms, the performance depends on both locality and load balance and these two metrics are often at odds with each other. Therefore, directly considering locality and load balance at the application level may make the application much more complex to program.

In this work, we develop locality-conscious concurrency platforms for multiple different structured parallel programming models, including streaming applications, task-graphs and PARALLEL_FOR loops. In all of this work, the idea is to minimally disrupt the application programming model so that the application developer is either unimpacted or must only provide high-level hints to the runtime system. The runtime system then schedules the application to provide good locality of access while, at the same time also providing good load balance. In particular, we address cache

locality for streaming applications through static partitioning and developed an extensible platform to execute partitioned streaming applications. For task-graphs, we extend a task-graph scheduling library to guide scheduling decisions towards better NUMA locality with the help of user-provided locality hints. CILKPLUS PARALLEL_FOR loops utilize a randomized dynamic scheduler to distribute work which, in many loop based applications, results in poor locality at all levels of the memory hierarchy. We address this issue with a novel PARALLEL_FOR loop implementation that can get good cache and NUMA locality while providing support to maintain good load balance dynamically.

Chapter 1

Introduction

In line with Moore's law, transistor counts have been steadily increasing over the years which, for a long time, allowed chip manufacturers to make faster processors. In recent years this increase in raw performance has not managed to keep up – despite having access to more transistors individual processors are no longer getting any faster. To address this, manufacturers began including multiple processing units (cores) on a single chip, ushering in an era of parallelism. Although a single core could not complete an application faster, if that application's workload could be divided up or shared, developers could continue to take advantage of new hardware to run their software faster. The recent emergence of commercial multi-processor systems, which combine multiple multi-core processors within the same machine, demonstrates the industry's intention to continue increasing the parallelism a system can support.

While some applications can easily take full advantage of the increasing number of cores when their workloads can naturally map to multiple cores, many can be challenging to implement correctly. Two of these programming challenges are particularly important determining factors when it comes to producing high-performance applications.

1. **Load-balance**: To get good load-balance, programmers need to ensure that all cores are performing roughly the same amount of work. Balancing the workload across the entire system ensures that no individual core is wasting time idling.

2. **Locality**: To get good locality, cores should do work which uses data that is *close by*. This means taking advantage of caches, re-using data when possible, and in the context of a multi-socket machine which employs non-uniform memory accesses, using data which is on the same socket. In either case, the goal is to mitigate the cost of accessing memory by ensuring the memory accesses can be as fast as possible.

It is often the case, however, that these two challenges are at odds with one another – a good solution for one may in fact be a bad solution for the other. When dealing with an application who's subtask workloads are not known ahead of time, often the best strategy is to dynamically assign subtasks to cores at runtime. Indeed many concurrency platforms employ randomized work-stealing, a provably efficient version of this dynamic scheduling. Dynamically scheduling subtasks might be great for getting good load-balance, but can be particularly poor at getting good locality. With no guarantee of what executes where, programmers are unable to ensure their application's subtasks use data that is close by, leading to a larger number of expensive memory accesses. To avoid this, programmers could instead distribute their application's data across the entire machine's memory and perform static scheduling to ensure the data needed by an individual subtask is close. This strategy can lead to optimal locality but at the cost of being highly sensitive to workload imbalances. Making scheduling decisions before-hand can result in a single core getting assigned more work than others.

The overall aim of this work is to bridge the gap between these two programming challenges and produce multi-core ready applications which get good-load balance while making decisions to improve locality. Solving these problems generally can be difficult as there are many different

application classes which require different strategies to be effective. To that end, we narrow our focus to two programming models and develop solutions for each individually.

1. **Streaming Applications**, where programmers connect subtasks with explicit communication channels and stream data between them. We investigate cache-conscious scheduling of streaming pipelines **??** in Chapters **??** and provide AMPIPE, a platform for developing parallel streaming applications in Chapter 4.

2. **Dynamic Multithreading**, where programmers denote opportunities for parallelism and allow the runtime system to make all assignment and scheduling decisions. In Chapter 5 we address locality-aware dynamic task-graph scheduling **??** and in Chapter 6 we look at a novel locality-conscious PARALLEL_FOR loop.

1.1 Streaming Applications

Streaming is an effective paradigm for parallelizing complex computations on large datasets. At a high level, streaming application can be described by a directed graph where nodes correspond to ***computation modules***, and edges represent directed FIFO channels between modules. The modules send data in the form of ***messages*** (also called ***tokens*** or ***items***) via these channels. In this work, we only consider ***pipeline*** topologies where modules are connected in a linear chain via channels. Scheduling individual computation modules on different cores allows these applications to execute in parallel.

For streaming applications, a schedule describes two things: where (on which core) does each iteration of each module execute; and the order of the execution of these iterations. In this work, we will primarily concern ourselves with ***static schedules*** — schedules where a module is placed on a particular processor and all iterations of that module execute on the same processor.

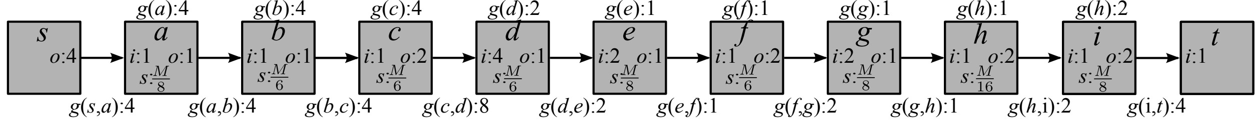


Figure 1.1: An example pipeline with annotated state size, input and output rates and gains for both modules and edges.

Arriving at a load-balanced static schedule for streaming applications involves finding an assignment of computation modules to cores such that the total workload across all cores is roughly equal. Such a scheduling decision can be modeled as a bin-packing problem.

Producing a streaming application that gets good locality is not as straight-forward. In this work, we focus on finding static schedules with good *cache* locality which reuse cached data as much as possible while reducing the number of accesses to data which does not reside in cache.

The streaming paradigm has been applied to diverse application domains such as media [42], signal processing [65], computational science [53] and data mining [34]. Several languages explicitly support streaming semantics, including Brook [22], StreamC/KernelC [41], StreamIt [71] and more recently RaftLib [11].

1.1.1 The Streaming Model

A streaming pipeline consists of a sequence of n ***computational modules***, and each module u has exactly one ***incoming channel*** (from the previous module in the sequence) and one ***outgoing channel*** (to the subsequent module in the sequence). The modules send data in the form of ***messages*** to each other via these channels. Channels may have ***buffers*** (implementing FIFO queues) to store messages that have not yet been consumed by the receiving module. We say that module u precedes module v, denoted by $u \prec v$, if u is before v in the sequence. We assume that the

incoming channel into ***source module*** s (the first module) streams an infinite amount of data into the pipeline and the outgoing channel from ***sink module*** t (the last module) streams it out.

Each module u has an associated state; we denote the size of this ***state*** by $s(u)$. In order to execute, or ***fire*** a module u on a processor, the entire state of that module must be loaded into that processor's cache. When the module fires, it consumes $in(u)$ data items from its incoming channel, performs some computation, and then produces $out(u)$ data items on its outgoing channel, where $in(u)$ and $out(u)$ are static parameters of the module. A module is ***ready*** to fire its input buffer contains at least $in(u)$ messages.

1.1.2 Assumptions and Definitions

Throughout this work, we make the several assumptions about the streaming pipeline — these assumptions are either necessary to admit any reasonable solution or are without loss of generality (made only to simplify the exposition). We assume that all messages are unit size and that the state of size each module is at most the size of the cache. The former assumption is without loss of generality given the arbitrary input and output rates. The latter is necessary to allow a module to be fully resident in cache when fired. In addition, we assume that the state size of each module also counts the minimum buffer required on its incoming and outgoing channels.

Finally, we define some terms and notation used throughout the work. We use the term ***gain*** to describe the rate of amplification of messages along the pipeline. In particular, The gain of a module is the number of times that module fires, on average, each time the source module consumes an input item. Therefore, the gain of the module is $gain(w) = \prod_{u \prec w}(out(u)/in(u)) \times \frac{1}{in(w)}$. The gain of edge (v, w) is the number of items produced on that edge, on average, each time an item is consumed by the source module, and is given by $gain(v, w) = gain(v) \times out(v)$. Figure 1.1 represents an example pipeline with module and edge gains computed.

We call a set of consecutive modules a ***segment***, and we denote the segment comprising modules between u and v (inclusive) by $\langle$u,v,. $\rangle$ In each segment s, we call the edge with minimum gain a ***gain-minimizing edge*** (if there is more than one, choose arbitrarily), denoted by $gainMin(s)$.

1.2 Dynamic Multithreading

In contrast with typical thread based applications, where the user explicitly defines parallel operations which are mapped to different threads, dynamic multithreading allows users to describe *opportunities* for parallelism and the decisions of what executes where is left up to runtime system. In this work we focus on Intel's CILKPLUS, an extension to C and C++ which offers such an environment. CILKPLUS adds to these languages the cilk_spawn and cilk_sync keywords which are handled by the provided runtime system. When the user issues cilk_spawn, it informs the runtime that the spawned work *can* execute in parallel with all of the code until the next cilk_sync. Applications of this type can be represented as a direct acyclic graph, where nodes represent instructions and edges represent the dependencies between them. Figure **??** demonstrates the type of graph a user could create with these new keywords.

Apart from offering an easy-to-use programming interface, CILKPLUS utilizes a provably good randomized runtime system which makes dynamic scheduling decisions to parallelize the user's application for them. By making scheduling decisions at runtime CILKPLUS is able to effectively load-balance these parallel applications resulting in good runtime performance.

Randomized scheduling decisions can, unfortunately, lead to applications with poor data-locality, especially in the context of systems with non-uniform memory accesses (NUMA). The user is incapable of structuring their application to ensure that the data necessary for an operation is within cache, or even within the same NUMA-domain. To resolve this, we augment these randomized

scheduling decisions to be **locality-conscious**, allowing the applications to still get good load balance and better data-locality.

We first accomplish this in a task-graph scheduling library built on top of CILKPLUS as well as within the runtime itself by a novel implementation We accomplish this balance, between load-balance and data-locality, in the domain of dynamic multithreading through two avenues:

1. a task-graph scheduling library built on top of the CILKPLUS runtime.

2. a novel implementation of PARALLEL_FOR loops to be provided by the runtime.

Chapter 2

Cache-Concious Scheduling of Streaming Pipelines

Cache efficiency is an important determinant in the performance of algorithms, and there has been extensive theoretical and practical work in designing cache-efficient algorithms for both sequential [1, 12] and parallel [16] machines. On a shared-memory machine, the communication cost roughly corresponds to a subset of the cache misses, but additional cache misses occur locally on each processor and are not reflected by communication cost. We are not aware of much prior work, particularly with a theoretical foundation, that considers the impact of cache effects on throughput in streaming applications. One example is from Agrawal et al. [6], who focus on designing cache-efficient scheduling of streaming applications on single-processor machines. Specifically, they show that for both pipelines and general acyclic graphs, a simple partitioning strategy generates a schedule that is asymptotically optimal for a single-level cache.

In this chapter, we consider the question of cache-conscious scheduling of streaming applications on parallel machines with private caches and find a static scheduling algorithm which minimizes cache misses.

2.1 Model

Our theoretical analysis is with respect to the parallel external memory (PEM) model [8], which is an extension of the external memory model [1] and similar to many other private-cache models in the literature (e.g., in [12]). The PEM model is a computational model consisting of P processors, each with a private cache of size M, and a global shared memory. The caches and shared memory are organized into ***blocks*** of B consecutive addresses. A processor can only read or write data in its own cache; when accessing data not in cache, a ***cache miss*** or ***I/O*** occurs, whereby the block must be loaded from shared memory to the processor's cache. The model allows a cache to store any M/B blocks simultaneously (i.e., caches are fully associative). All communication between processors occurs through the shared memory in the form of I/Os.

Processors may perform I/Os concurrently, which is called a ***parallel I/O***. The complexity measure is the number of parallel I/Os. Interpreted as time, accessing data in cache is free, but each I/O takes unit time. Thus in a single timestep, each processor may load up to 1 block or B elements, for a total of P blocks or PB elements. A "linear" I/O bound for, e.g., touching n elements in an array is $O(n/(PB))$.

Variants of the model specify when the same data may be resident in multiple caches, but these are not important for the present paper. We require exclusivity with respect to modules only — the same module may not be resident in two caches simultaneously. Otherwise, our lower bound applies to all model variants, and our upper bound does not have any data simultaneously loaded on any caches.

2.2 Theory

This section gives lower bounds on cache misses when scheduling a streaming pipeline on multiple processors, which we leverage later to prove that a partitioned schedule is optimal. The two bounds are analogous to lower bounds for other load-balancing problems. Specifically, we first lower-bound the total number of misses, which implies a lower bound on parallel I/Os (or time) since $\leq P$ occur concurrently. Our second bound addresses the case where parallelism P is not achievable due to local imbalance, i.e., if some small part of the pipeline dominates the running time. Combining the two gives a lower bound on running time that matches the upper bound achievable by a partitioned scheduler. Unfortunately, and surprisingly to us, the second lower bound is restricted to the case of a static scheduler and does not hold in general. We discuss this limitation at the end of the section, highlighting what makes proving bounds on (non-static) parallel streaming schedulers difficult.

2.2.1 Lower Bound on Total Misses

We first bound the total number of cache misses — the proof is inspired by the approach of Agrawal et al. [6], but with some changes to cope with the multiprocessor setting. The basic intuition is that any schedule (static or not) must "pay" for the messages crossing certain edges in the pipeline. The first lemma bounds the cache-miss cost for a single processor with respect to a single edge.

Lemma 2.2.1. *In the pipeline under consideration, let $s =$⟨u,v,b ⟩e any segment with total state size at least $2M$, and let $e = gainMin(s)$ be its gain minimizing edge. Any subschedule that fires module v at least $2M gain(v)/gain(e)$ times on a particular processor p incurs at least M/B cache misses on p. Moreover, these M/B cache misses are all due to either loading state from modules or messages on edges within the segment* ⟨*u,v,.* ⟩

Proof. The proof consists of two cases.

Case 1: Suppose processor p loads the entire segment ⟨u,v,d ⟩uring the subschedule. At most M of that state can already be resident in p's cache at the start of the subschedule, so p must incur at least M/B cache misses to complete the load.

Case 2: Suppose that some module in ⟨u,v,i ⟩s not fired by p during the subschedule. We define a message m to be a ***crossing ancestor*** if m is consumed by a module running on processor p during the subschedule, but m is not generated by a module on processor p during the subschedule. (If m was generated on processor p but *before* the subschedule began, it is still considered a crossing ancestor.) Since p does not fire the entire segment during the subschedule, all inputs to v during the subschedule are the progeny of some crossing ancestor. The number of crossing ancestors is minimized if they all occur at the gain minimizing edge e, and hence there must be at least $2M$ crossing ancestors in order to fire v a total of $2M gain(v)/gain(e)$ times. By definition, crossing ancestors are read by p, so each crossing ancestor must either already be resident in p's cache before the subschedule, or it must be loaded into p's cache during the subschedule. Since at most M crossing ancestors can be resident at the start of the subschedule, the remaining M crossing ancestors incur at least M/B cache misses. □

The following corollary combines 2.2.1 across all processors. The nuance here that necessitates the new lower bound, as opposed to applying Agrawal et al.'s bound [6] as a black box, is that P processors have PM cache in total instead of the M cache for the uniprocessor case.

Corollary 2.2.1. *Consider a streaming pipeline. Let $s =$⟨u,v,b ⟩e any segment with total size at least $2M$, and let $e = gainMin(s)$ be its gain-minimizing edge. Any subschedule that fires v at least $6PMgain(v)/gain(e)$ times in total across P processors must incur $\Omega(PM/B)$ cache misses. In other words, $\Omega((1/B)gain(e)/gain(v))$ is a lower bound on the amortized cost of firing v.*

Proof. From 2.2.1, if a particular processor p fires v a total of $f_p\lceil 2Mgain(v)/gain(e)\rceil$ times, then it incurs at least $\lfloor f_p\rfloor M/B$ cache misses. It remains to prove that $\sum_p \lfloor f_p \rfloor \geq P$, and hence the total number of cache miss is $\sum_p \lfloor f_p \rfloor M/B \geq PM/B$.

Since e is the gain-minimizing edge and $in(v) \leq M$, we have $gain(e) \leq gain(v) \cdot in(v)$ and thus $gain(v)/gain(e) \geq 1/in(v) \geq 1/M$. It follows that $2Mgain(v)/gain(e) \geq 2$, and hence $\lceil 2Mgain(v)/gain(e)\rceil \leq 3Mgain(v)/gain(e)$. Since there are at least $6PMgain(v)/gain(e)$ firings in total, we have $\sum_p f_p \geq 2P$ and hence $\sum_p \lfloor f_p \rfloor \geq P$. □

The following theorem combines the preceding corollary across the entire pipeline. Note that the theorem identifies which edges must be paid for with respect to an arbitrary segmentation of the pipeline (defined as "bandwidth" in [6]). Nevertheless, the bound holds for *any* schedule, even a non-partitioned and non-static schedule.

Theorem 2.2.2. *Consider a pipeline graph in which $S = \{\langle u_i, v_i\rangle\}$ is any collection of disjoint segments such that each segment has total size at least $2M$. Then for sufficiently large T, any parallel schedule of the pipeline that fires the sink node t at least $T \cdot gain(t)$ times must incur at least $\Omega((T/B)\sum_{s\in S} gain(gainMin(s)))$ cache misses in total.*

Proof. Observe that if t fires $Tgain(t)$ times, then v_i fires $Tgain(v_i)$ times. Thus if $T \geq 6PM/gain(gainMin(s))$ for $s \in S$, then we can apply 2.2.1 to s to get a cache-miss bound of $\Omega((T/B)gain(gainMin(s)))$. In addition, each application of 2.2.1 and 2.2.1 only counts misses for messages/state within each segment. Therefore, there is no double counting of cache misses. □

2.2.2 Lower Bound on Time

2.2.2 implies that the number of parallel I/Os is at least $\Omega((T/(PB)) \sum_{s \in S} gain(gainMin(s)),$ since at most P misses occur in any parallel I/O. This provides a lower bound on the running time of any schedule. We now argue that for *static schedules*, the gain-minimizing edge with the largest gain also provides a lower bound.

Theorem 2.2.3. *Consider a pipeline graph in which $S = \{\langle u_i, v_i \rangle\}$ is any collection of disjoint segments such that each segment has total size at least $2M$, and let t be the sink node. After t is fired at least $T \cdot gain(t)$ times for sufficiently large T, the running time is,*

- $\Omega((T/(PB)) \sum_{s \in S} gain(gainMin(s)))$ *parallel I/Os for any schedule, and*
- $\Omega((T/B) \max_{s \in S} gain(gainMin(s)))$ *parallel I/Os for any* static *schedule.*

Proof. The first statement follows from 2.2.2. For the second statement, consider segment $s =$⟨u,v,w ⟩hich has the largest gain-minimizing edge. 2.2.1 shows that for this segment, if the module v stays on one processor p, then p must incur an amortized $\Omega((1/B)gain(gainMin(s'))/gain(v))$ cache misses each time v is fired, each of which must be part of a different parallel I/O. □

Somewhat surprisingly, the second lower bound deriving from the maximum *does not* hold for general schedulers, only for static ones. We demonstrate by example: Suppose that the pipeline consists of $2kM/B$ modules, each of size B, with all gains equal to 1. Let $P = kM^2/B$ be the number of processors. For conciseness only, the following description assumes each processor has one extra block, i.e., $M + B$ cache size. Consider the following schedule: For each module u, assign a M/B processors, denoted P_u, to module u. Place cyclic buffers of size M^2 on each edge. Warm-up the schedule to get $M^2/2$ messages in each buffer, which is invariant between rounds in the remaining schedule. The schedule proceeds in rounds, in parallel on each module, consisting of a load step followed by an execute step.

Load step for u: foreach processor from P_u in parallel, load a distinct $M/(2B)$ full blocks from

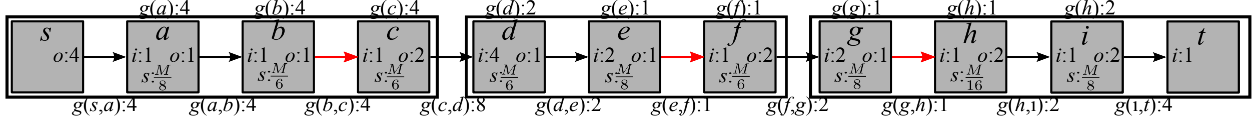
u's input buffer and also preload the corresponding $M/(2B)$ empty blocks from u's output buffer. These loads occur in parallel, so the total number of parallel I/Os is M/B.

Execute step for u: foreach processor from P_u but now sequentially (since u can not be fired on multiple processors at once), load u's state and fire it $M/2$ times. In the execute step, loading u incurs one cache miss on each processor for M/B parallel I/Os total, but the input and output buffers are already in cache, so firing u incurs no other I/Os.

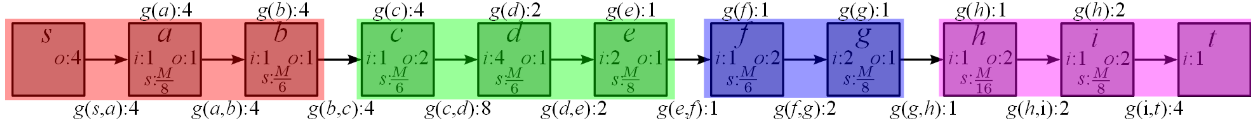
Putting it together, a single round takes $2M/B$ parallel I/Os but fires each module $M^2/(2B)$ times, for a total of $O(1/M)$ per firing. Thus if the sink fires T times, then the total time is $O(T/M)$.

This upper bound gives an example where the maximum-based lower bound (second condition in 2.2.3) does not hold. That bound would say that the total time is $\Omega(T/B)$ which is much larger than $O(T/M)$. This discrepancy arises from the fact that the example schedule moves modules across processors. The first condition of 2.2.3 still holds, since it only states that the lower bound on time is $\Omega(Tk/(PB)) = \Omega(T/M^2)$. Note that, setting $k = \Theta(B)$, this example only achieves a runtime of $O(Tk/(\sqrt{P}B)$ parallel I/Os, not $O(Tk/(PB))$. There is a tradeoff here; the example is able to beat the $\Omega(T/B)$ lower bound by increasing the total number of misses by a factor of M, which in turn increases running time arising from the total-work bound. Therefore, we have shown that a non-static schedule can do better than a static schedule, however, at the cost of increasing the total number of cache misses.

This section describes our cache-based partitioning algorithm, called *seg_cache*, for scheduling pipelines on multiple processors. For correctness, this algorithm assumes that the maximum state size of any module is at most $M/6$.



(a) A temporary segmentation found by greedily building segments up to size $\frac{M}{2}$ with the minimum gain edges highlighted.



(b) The final segmentation using the cross edges found from the temporary segmentation. Note that each segment has size at most M.

2.2.3 The *seg_cache* Algorithm

The algorithm produces a partitioning by first dividing the pipeline into temporary segments $S = \{s_1, s_2, \ldots, s_k\}$, where each segment $s_i = \langle u_i, v_i \rangle$ for $i < k$ has total state between $M/3$ and $M/2$ — the last segment s_k may be smaller. This temporary segmentation can be selected greedily: Initially create one segment s_1 as the current segment. Iterate over the modules in order. Add the current module to the current segment. If the total state of the current segment s_i exceeds $M/3$, create a new segment s_{i+1} and set that to be the current segment. Under the assumption that modules have state at most $M/6$, this process produces segments with total state at most $M/3 + M/6 = M/2$. Figure 2.1a shows this temporary segmentation for the pipeline in Figure 1.1.

Then select the minimum gain edge $gainMin(s_i)$ within each segment $s_1, \ldots, s_{k-1}$, with the exception of the last segment s_k. We call these edges the ***cross edges***, as they are the edges crossing between our final segments. That is, our final segmentation $R = \{\langle x_i, y_i \rangle\}$ is the one induced by cutting the selected cross edges. Since each final segment spans at most two temporary segments, each of size at most $M/2$, each segment in R has total state at most M. Figure 2.1b shows this final segmentation — note that we cut the gain minimizing edge in each of the temporary segments. If there are two edges with the same gain, we can choose arbitrarily.

We next load balance the set of segments R across processors as follows. We define the ***load*** of a segment $r = \langle x_i, y_i \rangle$ as $load(r) = gain(x_i)in(x_i) + gain(y_i)out(y_i)$, i.e., the sum of the gains of the incoming edge and the outgoing edge. Our load-balancing also employs a simple greedy strategy. Specifically, calculate the total load $L = \sum_{r \in R} load(r)$. The average load across P processors would thus be L/P. In pipeline order, greedily place segments from R on the current processor until the total load on the processors exceeds L/P, and then move to the next processor. The in-order aspect of this greedy load balancing guarantees that each processor is assigned a contiguous set of segments. Therefore, for the example shown in Figure 2.1b, if we had 2 processors, the first segment on the first processor and the last three segments on the second processor.

It remains to define buffers on cross edges and the actual schedule. To simplify the description and analysis, we describe a version that schedules in synchronized rounds with a large period and correspondingly large buffers. To calculate the buffers on cross edges and the period, we choose the value $X = M \prod_{\text{modules } i} in(i) \cdot out(i)$. In this way, for every edge e (and in particular for cross edges), we have that $Xgain(e)$ is an integer larger than M. We add buffers of size $Xgain(c)$ to every cross edge c. Our actual implementation (which is also synchronous) does not employ buffers this large and the buffer sizes can be further reduced by scheduling asynchronously.

The schedule itself is periodic and divided into rounds, with synchronization between rounds. In each round, each processor loads each of its segments exactly once in pipeline order; once a segment is loaded, the contained modules are fired many times. Specifically, for $\langle x_i, y_i \rangle$, if the input buffer is full — that is, there are $Xgain(x_i)in(x_i)$ messages available on the incoming edge — the segment is ready and is run.[1] To run the segment $\langle x_i, y_i \rangle$, execute the latest module within the segment with enough inputs available to fire. Repeat until y_i has fired $Xgain(y_i)$ times. By

[1]Otherwise, wait until the next round. This waiting occurs only in the earliest rounds until a steady state is reached — the ith processor first becomes ready in the ith round.

careful construction of X, all modules $j \in \langle x_i, y_i \rangle$ have fired $Xgain(j)$ times, and hence there are no messages on any internal edges, and the subsequent segment becomes ready.

2.2.4 Bounding *seg_cache*'s Performance

We now prove an upper bound on the number of cache misses incurred on each processor with respect to its load, which is defined as follows. Let R_p be the set of segments assigned to p by *seg_cache*. Then the ***load of processor p*** is defined as $procLoad(p) = \sum_{r \in R_p} load(r)$.

Lemma 2.2.4. *In each round of* seg_cache*, the total number of cache misses incurred by* p *is* $O((X/B) procLoad(p))$.

Proof. When executing a segment $r = \langle x, y \rangle$, its entire state is loaded just once, since it fits in cache. (By assumption, the state includes a small buffers on internal edges to accommodate varying input/output rates.) Thus when running r, the only cache misses are from loading the state initially, and reading/writing messages from/to the incoming/outgoing cross edges. The state load costs $O(M/B)$. The cost on cross edges is $O(Xgain(x)in(x)/B)$ and $O(Xgain(y)out(y)/B)$, respectively, which sum to $O((X/B)load(r))$. Since $Xgain(e) \geq M$ for all edges e, $O((X/B)load(r))$ dominates. Summing over all segments r assigned to p completes the proof. □

We obtain the following corollary on time by taking the max of Lemma 2.2.4 across all processors.

Corollary 2.2.2. *The duration of each round is* $O((X/B) \max_{\textit{proc. } p} procLoad(p))$ *parallel I/Os.*

We now upper-bound the running time of *seg_cache*.

Lemma 2.2.5. *For sufficiently large* T*, the time required for* seg_cache *to fire the sink node* t *a total of* $Tgain(t)$ *times is* $O((T/B) \max_{\textit{proc. } p} procLoad(p))$ *parallel I/Os.*

Proof. If a segment fires X times its gain times each time it becomes ready, then a simple inductive argument shows that all the segments on processor i first become ready in round i. Moreover, once ready, they continue to become ready again in each subsequent round. Thus, after Z rounds, the last processor's segments run during at least $Z - P$ of them, and hence t fires at least $(Z - P)X gain(t)$ times. From Corollary 2.2.2, the total time required for Z rounds is $O(Z \cdot (X/B) \max_{\text{proc. } p} procLoad(p))$. As long as $T \geq PX$, choosing $Z = \lceil 2T/X \rceil$ completes the proof. □

The following theorem states that *seg_cache* is asymptotically optimal, when given a constant factor memory augmentation. Note that in the theorem statement, $procLoad(p)$ is a metric of the pipeline — we are not just saying that *seg_cache* is optimal for this particular static schedule, but rather that no other static schedule is much better.

Theorem 2.2.6. *Let $procLoad(p)$ denote the processor loads arising from an execution of* seg_cache *on the pipeline. Then for sufficiently large T, every static schedule on a machine with cache size of $M/6$ requires a runtime of $\Omega((T/B) \max_{proc.\ p} procLoad(p))$ parallel I/Os to fire the sink node t a total of $T gain(t)$ times.*

Proof. Consider the temporary segmentation S chosen by *seg_cache*, and let C be the set of gain-minimizing cross edges selected. Each of the segments (except the last which contributes no cross edge) has size at least $2(M/6)$, and thus applying 2.2.3 for a machine with $M/6$ allows us to conclude that $\Omega((T/(PB)) \sum_{c \in C} gain(c) + (T/B) \max_{c \in C} gain(c))$ is a lower bound on the runtime of any static scheduler.

We next show $(1/P) \sum_{c \in C} gain(c) + \max_{c \in C} gain(c)) \geq (1/2) \max_{proc.\ p} procLoad(p)$, which completes the proof. Cross edges contributes to the load of two segments, so the total load is $L = 2 \sum_{c \in C} gain(c)$. The *seg_cache* algorithm adds at most one segment to a processor after it reaches L/P load. Each segment borders two cross edges, so the load of any segment is at

most $2 \max_{c \in C} gain(c)$. Thus the maximum load on any processor is $\max_{proc.\ p} procLoad(p) \leq (2/P) \sum_{c \in C} gain(c) + 2 \max_{c \in C} gain(c)$. $\square$

2.3 Conclusions and Future Work

In this chapter, we studied cache-conscious scheduling of streaming pipelines on machines with private caches. We've presented *seg_cache* which finds static schedules for streaming pipelines by performing a gain-minimizing partitioning.

Although we focused on reducing misses in the last-level private cache, we could extend our algorithm to minimize misses at all levels of the memory hierarchy. A natural progression for future work would be to deploy streaming pipelines on NUMA machines while reducing the number of remote memory accesses.

Another avenue for future work would be expanding our algorithms to support different classes of streaming applications. Allowing for more expressive structures like series-parallel or general directed acyclic graphs would enable us to handle a larger range of application types.

Chapter 3

Executing Streaming Pipelines

In this Chapter, we evaluate the performance of our cache-based scheduling algorithm *seg_cache* against a number of other viable static scheduling algorithms for streaming pipelines. Although the direct aim of *seg_cache* is to reduce cache misses, our final goal is to produce applications with higher throughput which processes a fixed set of input as fast as possible.

To perform this evaluation, we constructed an executing environment to emulate the execution of randomly generated streaming pipelines. The environment takes in: 1) a pipeline definition 2) a scheduling strategy, and 3) the amount of input to process. Our environment emulates this execution on real hardware. By comparing multiple scheduling strategies on the same pipeline, we're able to directly compare the impact of each strategy on the resulting performance.

3.1 Scheduling Policies

We evaluated several static schedulers. Each schedule is characterized by 3 features: (1) The segmentation. We describe all of our schedules in terms of segments — a set of consecutive modules restricted to the same processor. Note that a segment may contain anything between one module and all the modules. How segments are chosen varies by scheduling policy. (2)

The processor assignment. Each schedule defines how the segments are (statically) assigned to processors. Multiple segments may be assigned to the same processor. (3) Cross-edge buffer allocation. Each scheduling policy also defines the size of buffers to place on cross edges — the edges that go between segments.

All the other details for all policies are similar. For any internal edge e from module u to module v of the same segment the buffer of size $2\text{lcm}(out(u), in(v))$ — that is the least common multiple of the number of items produced and consumed on the edge — is allocated.[2] For all policies except *seg_cache*, the cross edge buffers are also assigned in the same manner. In all the computation-based policies (*seg_runtime*, *bin_full* and *bin_empty*), before load-balancing, we normalize the modules computation cost based on the gain of the module. The scheduling policies compared in this section are defined as follows:

1. *seg_cache*: Segmentation based on cache (our policy) as described in 2.2.3. Each cross edge e has a buffer of $(M/2)gain(e)$.

2. *random_assignment*: Segments consists of single modules and are assigned to processors at random.

3. *seg_random*: Random segmentation. A single segment is created per processor by randomly picking $p - 1$ edges to be cross edges.

4. *seg_runtime*: Segmentation based on computation cost. In this policy a single segment is created per processor minimizing the maximum number of compute steps on a single segment by using a greedy load-balancing policy analogous to that of *seg_cache*.

5. *bin_full*: Fullest first binpacking on computation. Assign modules to a segment until their cumulative computation is just smaller than $1/P$ fraction of the total computation, creating

[2]This internal buffer could be sized $out(i) + in(i + 1)$, but the larger quantity allows for easier scheduling and does not affect the performance.

P large segments, one for each processor. The remaining modules are in their own segments and assigned greedily to the processor with the smallest overall compute cost. This policy often results in a segmentation nearly identical to *seg_runtime*.

6. *bin_emptiest*: Emptiest first binpacking on computation. Each segment consists of a single module. We greedily assign segments to the processors with the fewest total compute steps.

3.2 Pipeline Execution

We built an execution engine that takes in a pipeline and the static-schedule description and executes the pipeline according to that schedule. The execution engine runs exactly one thread on each processor. The ***master thread*** begins execution and is responsible for parsing the schedule and spawning all the remaining ***worker threads*** and pinning them on their designated processors. These worker threads are responsible for executing the segments assigned to respective processors. The master thread executes the source node, generating all the subsequent messages to be processed, and the sink node, consuming all the messages after processing.

In a loop, each worker thread finds a segment that is ***ready*** — the buffer on the input cross edge is at least half-full and the buffer and the output cross edge is at least half-empty. This segment is then loaded into memory. At this point, this segment is executed until either the buffer on the input cross edge is empty or the buffer on the output cross edge is full. If there are multiple ready segments, the worker thread always executes the last (in pipeline order) ready segment. Once a ready segment is picked for execution, it is executed until either the buffer on its incoming cross edge is empty or the buffer on its outgoing cross edge is full. When a segment is executed, its internal modules are fired one at a time. Any schedule of firing its internal modules is valid — in our execution engine, the last ready module (in order of the pipeline) is always fired.

The internal-edge buffers between modules within a segment are implemented as simple FIFO queues. No synchronization is required on internal edges because all modules within a segment are consecutive and are scheduled on the same processor. Neighboring segments can, however, be scheduled on different processors, and we thus use thread-safe single-producer single-consumer lock free ring buffers for communication through cross edges between segments

3.3 Experiments

We now experimentally compare the running time of our strategy against several other static policies when executing randomly generated pipelines drawn from distributions with varying characteristics. The experiments indicate that under many conditions, cache-based segmentation is able to improve the overall running time, while under other conditions, computation time based partitioning performs better. We also investigate a heuristic that combines computation and cache misses as a basis of partitioning, and find that it often provides the best of both worlds.

3.3.1 Pipeline Generation

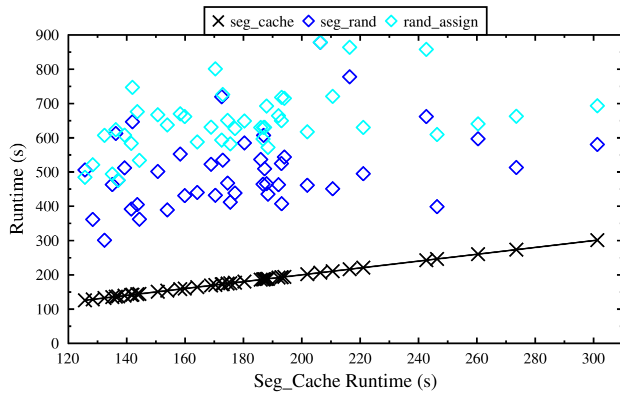
Our experimental evaluation consists of running each schedule on randomly generated pipelines. We fixed the total number of modules to 140 in order to allow for sufficient differences in the schedules generated. For each pipeline, we randomly generate three sets of parameters: (1) gain of each edge, (2) state size of each module, and (3) computation requirement of each module. The gain of each edge is picked randomly between $[1, M/16]$, where M is the size of the cache. The upper limit is picked so that the maximum gain is pretty large, but still allows all non-cross edge buffers to fit in memory. The state size of each module is picked randomly between $[1KB, 32KB]$,

again in order to allow the each module to individually fit in cache. Finally, the module computation times are selected from $[0.5\mu s, 50\mu s]$. We used two kinds of distributions for generating our parameters — namely ***uniform distribution*** and ***zipf's distribution***, [76]. Zipf's is a heavy tailed distribution with a probability density is given by $P(x) = \frac{x^{-p}}{\zeta(p)}$. Here ζ is the Riemann zeta function and p is a positive parameter which controls how heavy the tail is. For our experiments we use $p = 1.5$

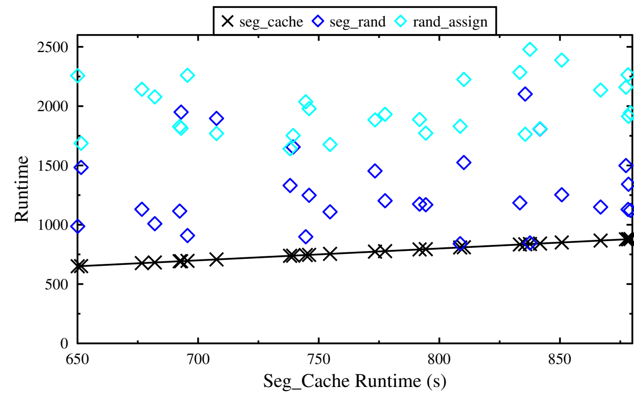
3.3.2 Experimental Settings

All of our experiments were run on Intel Eight Core Xeon E5-4620 2.2Ghz processors with 8-way set associative L2 caches of $M = 256$KB and block size $B = 64$ bytes. Each message is the size of a word (4 bytes) and the source node generates $16384M$ ($16384 \times 1024 = 16777216$) messages in total during each execution. This large number ensures that the overhead of loading and clearing the pipeline is low compared to the total execution time — therefore the total execution time is a reasonable approximation of the steady-state throughput.

Each chart consists of a set of trials using the same parameters to randomly generate the pipelines. In all charts, the y-axis is running time — lower is better. A vertically aligned set of points plot the running times of each scheduler on the same pipeline. The x-axis is the running time of *seg_cache*. By construction, the running time of *seg_cache* fits a line, which is shown. If most of the points for a particular strategy fall above the line, then that strategy is generally worse than cache-based segmentation; if the points generally fall below the line, then it is better than cache-based segmentation.



(a) Zipf gain, uniform state, uniform computation



(b) Uniform gain, uniform state, uniform computation

Figure 3.1: Normalized results for *seg_rand* and *rand_assign* against *seg_cache* with computation simulation disabled. We see that *seg_cache* works better with zipf gain than with uniform gain.

3.3.3 Disabled Computation

We first conduct a simple experiment to see if cache performance has any effect on runtime. We set the computation time for all modules to 0 and compare *seg_random* and *rand_assign* scheduling policies against *seg_cache* (the other policies depend on computation time).

Figure 3.1a shows the results for pipelines that have been generated using a zipf distribution for the gains and a uniform distribution for state size. We see that *seg_random* generally requires at least twice as much time to complete as *seg_cache*. While *seg_random* is able to take advantage of the temporal locality arising from segmentation in general, a random selection does not properly reduce the cost on cross edges, resulting in some bottlenecked processor. Since *rand_assign* lacks this temporal locality, it performs even worse. In Figure 3.1b we see that when a uniform distribution is used for gain selection *seg_random*'s performance is much closer to *seg_cache*'s. This is due to the fact that *seg_cache* concentrates on cutting low gain edges; with uniform distribution, there is little difference between a "good edge" (low gain edge) and a random edge. Therefore, the primary advantage of *seg_cache* is reduced compared to *seg_random*. Due to the lack of temporal

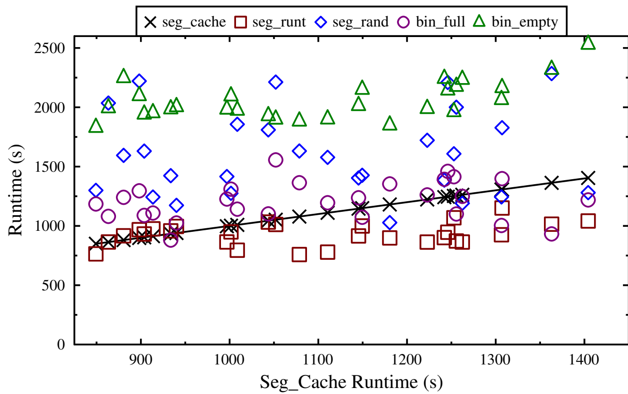
locality, *random_assign* still performs much worse. In subsequent experiments, we will omit the *random_assign* policy since it almost always much worse than the others.

This experiment provides evidence that cache performance of scheduling policy can play a role in determining the running time. In addition, a uniform distribution on the gains decreases the effectiveness of a cache-based segmentation due to the increased number of expensive edges which makes it difficult to pick "cheap" cross edges. Zipf's like heavy-tailed distributions commonly occur in practice in many circumstances and some studies [39, 40, 48] claim that the sizes of Unix jobs follow a heavy-tailed distribution. Therefore, it seemed reasonable to try these distributions in our experiments.

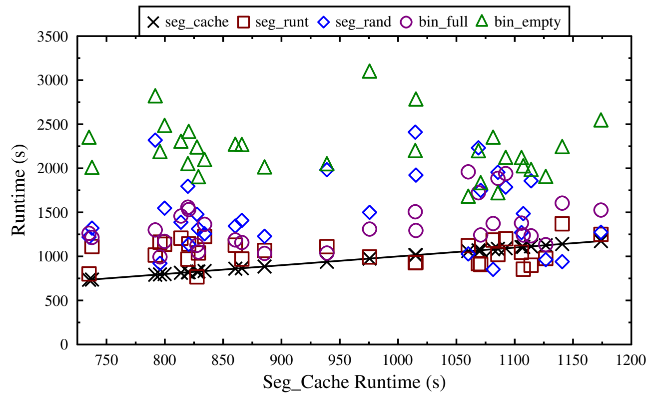
3.3.4 Randomly Generated Computation Times

The remainder of the experiments will consider computation times. Figure 3.2 shows the results when state sizes are selected uniformly at random, but the distribution of the gain and computation varies. As in the case where there is no computation, we again see, in Figures 3.2a and 3.2b, that uniform gains are not particularly good for *seg_cache* and computation-based segmentation *seg_runt* performs as well or better. The reason is similar as in the case of disabled computation — when the gains are uniformly distributed, *seg_cache* has very little ability to pick better cross edges (smaller gain cross edges) than other policies.

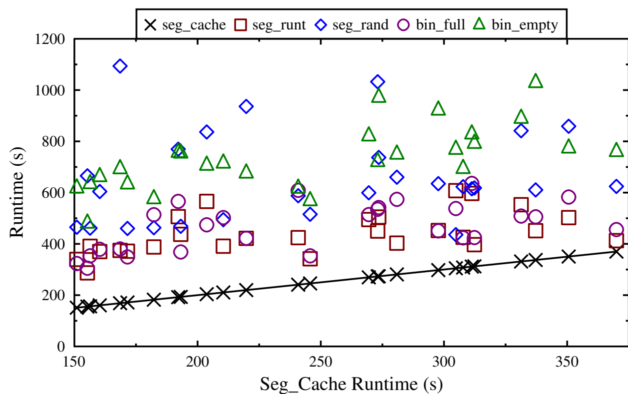
We see another interesting effect here — *seg_runt* is better with respect to *seg_cache* when computation is distributed uniformly (Figure 3.2a), while it is comparable to *seg_cache* when computation is generated using the zipf's distribution (Figure 3.2b). This is due to the difference in distributions: Our uniform distribution has a higher mean than the zipf distribution; therefore, the overall runtime is larger with the uniform distribution than it is with the zipf's distribution. Therefore,



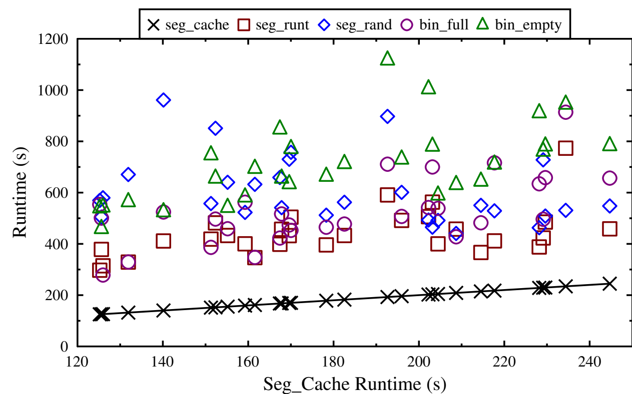
(a) Uniform Gain, Uniform State, Uniform Computation



(b) Uniform Gain, Uniform State, Zipf's Computation



(c) Zipf's Gain, Uniform State, Uniform Computation



(d) Zipf's Gain, Uniform State, Zipf's Computation

Figure 3.2: Normalized results for *seg_runtime*, *seg_random*, *bin_fullest* and *bin_emptiest* against *seg_cache* with computation time enabled. We see that *seg_cache* performs the best when both the computation time and the gains have a zipf's distribution. This is due to the fact that (1) zipf's distribution on computation time reduces the total computation time of the pipeline, making it more likely that the runtime is dominated by cache misses and (2) zipf's distribution on gains allows *seg_cache* more leeway to pick better cross edges.

computation cost is more likely to dominate the performance with uniform distribution on computation and hence, a policy that load-balances computation performs better. On the other hand, under the zipf's distribution, the computation of most modules is small; therefore, the overall computation is small and the cache effects can become more prominent and *seg_cache* can start to have some advantage.

In comparison, (just as with disable computation) when the gains are zipf's distributed, in Figures 3.2c and 3.2d, we see that *seg_cache* generally outperforms the other policies. In particular, even when the computation is uniformly distributed (Figure 3.2c) the ability to pick good cross edges allows *seg_cache* to perform better than other policies. Again we see in Figure 3.2d that when computation time is selected with a zipf distribution, cache performance plays an even higher role and *seg_cache* performs the best.

We ran other combinations of distributions (omitted due to space constraints) and these general trends seem to hold. The state-size distribution generally does not seem to have an appreciable effect on the relative performance of policies.

3.3.5 Correlated Computation Time

Thus far our experiments have selected all gains, state sizes and computation times independently. It is, however, often the case in practice that state size and computation time are correlated. Figure 3.3 shows the results when we pick the state s using the uniform distribution, and then the computation time of the module is its $(s/maxState) \times maxComp$ where $maxState = $ 32KB and $maxComp = 50\mu s$. Again we see that *seg_cache* outperforms all scheduling policies in the majority of cases. We see that the results are quite similar to Figure 3.2c, which is not surprising since again state size and computation are uniformly distributed. In fact, the distribution on computation times is not identical; the range of computation times for the un-correlated experiment shown

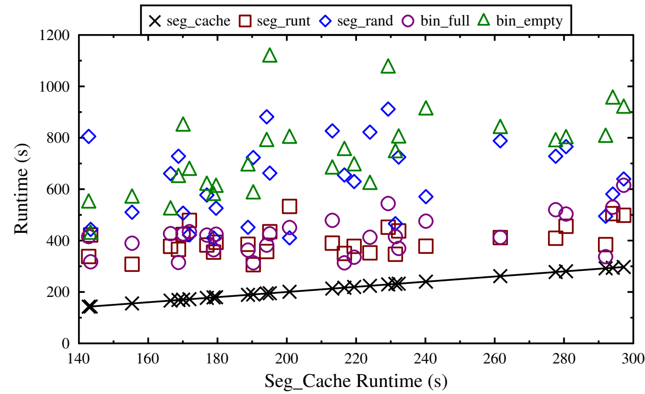
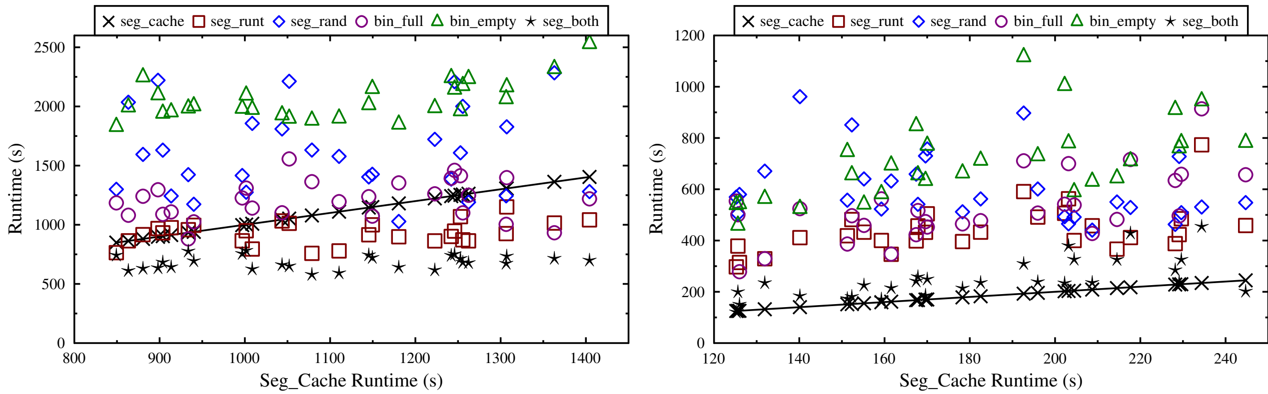


Figure 3.3: *seg_runtime*, *seg_random*, *bin_fullest* and *bin_emptiest* against *seg_cache* for correlated state size and computation time.

(Figure 3.2c) is $[0.5\mu s, 50\mu s]$ while for the correlated experiment it is $[1.56\mu s, 50\mu s]$. Therefore, the overall computation time of these correlated pipelines generally larger and therefore, we would expect *seg_cache* to perform worse. However, since *seg_cache* balances the state sizes in its segmentation, due to the correlation, in this case, it appears to also manage to balance the computation times to a certain extent, leading it to perform better than other strategies.

3.3.6 Segmentation Based on Computation Time and Cache

These experiments indicate that segmentation is good, since either cache-based segmentation (*seg_-cache*) or computation-based segmentation (*seg_runt*) generally dominates the other policies. Therefore, we experimented with a new algorithm, *seg_both*: a segmentation strategy that considers both computation and cache misses. For this policy, we first estimate the time for a cache miss (we found it to be about 3 nanoseconds). We then calculate the normalized load for all possible segments $\langle$x, y,t $\rangle$hat fit in cache by including both the time spent incurring cache misses and the time spent on

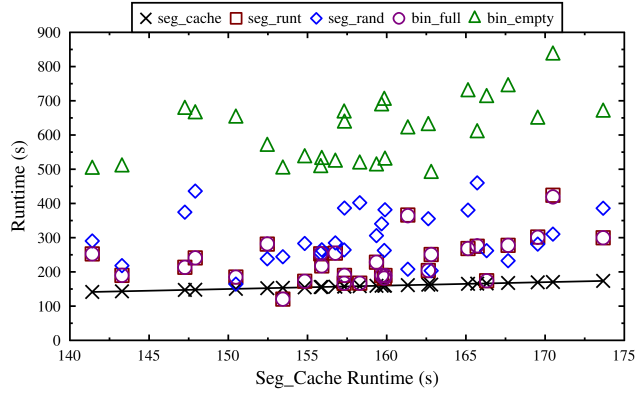


(a) Uniform Gain, Uniform State, Uniform Computation (b) Zipf's Gain, Uniform State, Zipf's Computation

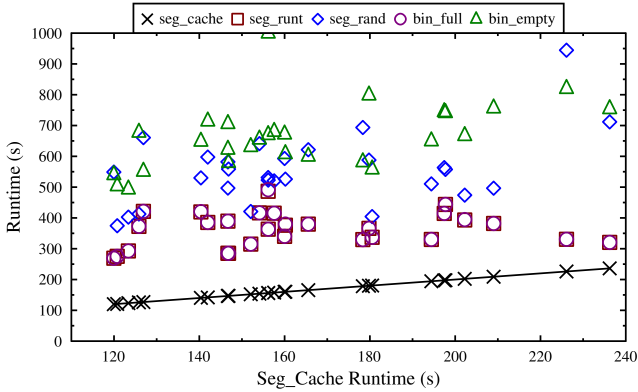
Figure 3.4: Segmentation based on both computation time and cache. We see that this form of segmentation does a better job of load-balancing computations when computation-based strategies dominate, but also gets most of the advantage of cache-based segmentation when *seg_cache* dominates.

modules' computation. We then greedily assign contiguous segments to processors so as to minimize the maximum load per core via a binary search algorithm. Since this algorithm considers cache misses, the buffers on cross edges are assigned in the same manner as *seg_cache*.

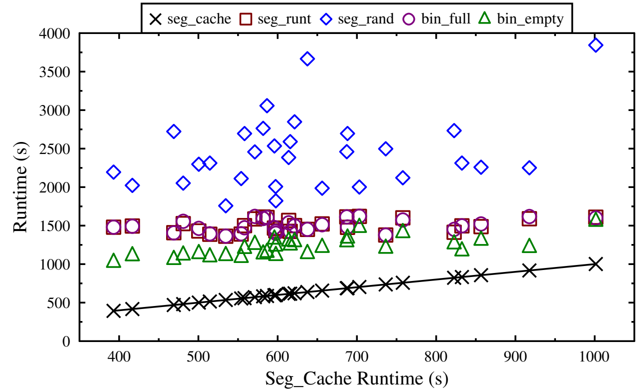
Figure 3.4 shows the results for this policy for both the case of uniform gain, uniform computation and for zipf gain, zipf's computation. If we compare Figure 3.4a to Figure 3.2a, we see that in the case where computation-based segmentation is better than cache-based segmentation, this combined segmentation policy beats both *seg_cache* and *seg_runt*. On the other hand, comparing Figure 3.4b to Figure 3.2d indicates that when cache-based segmentation is better, this combined segmentation does almost as well as *seg_cache*. Therefore, this policy seems to provide the best of both worlds.



(a) Uniform Gain, Small State and Computation



(b) Uniform Gain, Med. State and Computation



(c) Uniform Gain, Large State and Computation

Figure 3.5: Normalized results *seg_runtime*, *seg_random*, *bin_fullest* and *bin_emptiest* against *seg_cache* when state size and computation times have been fixed and gains are selected with a zipf distribution.

3.3.7 Fixed State and Computation

In order to better understand these scheduling policies, we conducted experiments where the module state sizes and computation times have been fixed and only the gains vary. Figure 3.5 shows three experiments, using 'small','medium' or 'large' values, where $state \in \{\frac{M}{1024}, \frac{M}{16}, \frac{M}{2}\}$ and $computation \in \{1, 10, 50\}\mu s$. In all experiments, *seg_cache* performs as well as or better than the other policies, since balancing the state sizes automatically balances the computation as well.

In Figure 3.5a both *bin_fullest* and *seg_runtime* are able to perform nearly as well as *seg_cache*. Note that the total state of the entire pipeline is small. Since both these policies create approximately 1 segment per processor and these segments fit in cache. Therefore, these policies get most of the advantage of *seg_cache* since they never have to pay to reload the state and only pay for 2 cross edges per processor. *bin_empty* is significantly worse since it has too many cross edges. As we increase the state sizes, the modules assigned to each processor no longer fit entirely in cache and the cost of loading segments starts mattering.

Another interesting trend is that *bin_empty* starts performing better as the computation and state size increase. The reason for this is subtle. Since *seg_runt* and *bin_full* essentially perform segmentation based on computation cost, they put large contiguous segments on each processor as the state size and computation increases. At the largest size, each processor's cache can only fit 1 or 2 modules at a time. Therefore, both these policies load a single module, execute it, and then load the next module, and so on, paying a large cache cost for each firing. On the other hand, *bin_empty* distributes contiguous modules across processors with each segment consisting of a single module — therefore, for large state sizes, it is essentially doing what *seg_cache* does, but without the advantage of large buffers on cross edges. This policy potentially allows it to execute each module many times before loading the next module.

3.4 Related Work

Most existing work on scheduling streaming computations computation costs of the modules only and tries to balance to computation across processors in order to maximize throughput. Bokhari solved the throughput optimization problem for pipeline mapping by finding a minimum bottleneck path in a layered graph that contains all information about application modules [17]. Hansen et al. later improved Bokhari's solution using dynamic programming [38]. Subsequently, many efficient algorithms have been proposed, both for homogenous processors [29, 59, 62] and heterogenous processors [15]. Researchers have also considered distributed memory communication models of various kinds to understand the effect of communication on throughput of pipeline computations [2, 3]. Researcher have also considered the problem of maximizing throughput and minimizing latency as a bi-criteria scheduling problem [13, 14]. In addition, replication of state-less modules in order to improve throughput has also been studied [30,44,50,68]. None of this research explicitly addresses the problem of minimizing the number of cache misses.

Heuristic cache-aware scheduling of streaming programs on both single processors and multiprocessors has been studied by several research groups [43, 56, 66]. Since all of these algorithms are based on heuristics, they do not guarantee optimality with respect to the number of cache misses. However, their empirical results support our claim that optimizing the number of cache misses can lead to significant improvement in performance. The only prior theoretical work that considers the number of cache misses for streaming applications considers only single processor schedules in both the cache-aware [6] and cache-oblivious [5] setting.

3.5 Conclusions and Future Work

In this Chapter, we evaluated the impact of scheduling policies on the performance of streaming pipeline applications. In most of the experimental benchmarks, scheduling based solely on the modeled cache effects is indeed significantly more effective than the more common load-balancing of computation costs. In the case that computation cost dominates the workload, a mixed strategy that takes into account both cache misses and computation cost outperforms one designed to balance the computation. We conclude that the cost of cache misses should not be ignored when designing scheduling strategies for streaming pipelines.

In Chapter 4 we generalize our execution engine and investigate how our scheduling policies impact real-world applications.

Chapter 4

AMPipe: Automatically Mapping Streaming Pipelines

4.1 Intro

In Chapter 2 we explored static scheduling strategies for streaming pipeline applications on parallel machines with private caches. We presented a prototype system in Section **??** which utilized generic configurable kernels which allowed us to test on a range of pipeline applications types with varying state, runtime and edge gain properties. While these experiments effectively captured a number of classes of applications, they were nevertheless synthetic and could not perfectly represent the real-world. To that end, our aim in this chapter is to integrate the mapping and scheduling systems into a full streaming application platform allowing us to implement real-world applications for experimentation.

We thus present an extensible framework, AMPIPE, for developing streaming pipeline applications. AMPIPE evolved as an extension of SCALAPIPE which allows users to easily develop streaming-pipeline applications for heterogeneous systems where different kinds of hardware may be used for individual computation kernels. AMPIPE's focus is instead on applications targeting

chip-multi-processors and it therefore has a different set of design goals. The programmers simply provide all the modules of the pipeline and their connecting edges but they do not have to specify the details of the schedule such as which module(s) run on which processor and how the modules communicate with each other. The framework provides a number of the mapping algorithms described in Chapter 2 which automatically decide which module to schedule on which processor. In addition, AMPIPE provides the runtime scheduler used in our prototype that automates the firing of modules and orchestrates the communication between them. AMPIPE is designed to be an easy-to-use, extensible streaming-pipeline system which produces parallel applications without requiring users to be concerned with traditionally difficult tasks like synchronization, communication or deadlock.

4.1.1 Contributions

AMPIPE has the following features:

Programming interface and profiling: AMPIPE provides a simple programming interface where the user can simply specify each module by specifying its computation; the number of elements it reads and writes in its input and output edges each time it fires; and the types of these elements. In addition, the programmer separately specifies the pipeline topology by connecting these modules. Therefore, the same module may be used in various pipelines easily. In order to create good mappings, AMPIPE requires some additional information about kernels. AMPIPE also provides profilers to allow users to collect this information about each module easily.

Automatic Mapping and Replication: Given a pipeline, AMPIPE provides automatic mapping algorithms that try to maximize the throughput of the computations. In addition, users can also write their own mapping algorithms easily in AMPIPE. We now describe a couple of automatic

mappers AMPIPE provides with an example; while there are other mapping algorithms provided with AMPIPE, we find that these perform the best.

Consider the pipeline in Figure 4.1a with 6 modules that we want to schedule on 4 processors. The number in each kernel shows the normalized computation requirement of that kernel and the numbers on the edges represent the normalized data volume that crosses this edge on average. The first, and simplest, mapping algorithm, called **SEGRUNTIME**, simply tries to load-balance the modules across all processors based on their computation requirements — the mapping is shown on the Figure 4.1b for 4 processors. Each shaded region executes on a single processor. In this case, The first, second and third modules execute on their own processor while the last three modules execute on the last processor. However, this mapper does not take the cost of cache misses into consideration at all. Note that the second edge carries more data than all the other edges. With the mapping created by SEGRUNTIME, all the data on this edge causes cache misses on private caches since it must be sent from the first processor to the second processor. The next mapping algorithm, **SEGBOTH**, takes both computation and cache misses into consideration and for this example it creates the mapping shown in 4.1c — note that it puts the first and second module on the same processor in order to ensure that the high-volume edge is "not cut" and the data on this edge stays in cache.

Note that both of these mappings are quite load-imbalanced since module C is the bottleneck module and they can only use one processor to work on this module; therefore, all the other processor will spend a bulk of their time idling waiting for the processor running this module to produce/consume data. In order to solve this, we incorporate ***replication*** in both SEGRUNTIME and SEGBOTH. Figure 4.1d shows the mapping we add replication to SEGBOTH. Here the bottleneck module has two copies, and each runs on different processor. Half the data is processed on each copy making the average load of each copy 6. Not all modules are replicable — AMPIPE allows users to

specify which modules are replicable and then automatically finds the best mapping considering replication.

Runtime Scheduling, Coordination and Communication: One the mapping is created, AMPIPE generates code for all the communication and synchronization between modules. For instance, the streaming abstraction assumes that each edge has an infinite buffer; however on a real platform, we must use finite buffer sizes. AMPIPE automatically uses appropriate buffer sized based on the situation. The edges between modules that execute on the same processor have small buffer sizes to both conserve space and ensure that all the data on these edges remains in cache. On the other hand, the edges that go between processors have larger buffer sizes (based on the mapping algorithm) to reduce idle time. In addition, AMPIPE also has 5 types of edges and picks the correct type of edge based on the situation. In particular, if we have an edge where the source and the destination modules are on different processors, then AMPIPE uses a thread safe ring buffer. More interestingly, AMPIPE automatically orchestrates communication for replication. Note that in the presence of replication, the non-replicated modules should still see data in the correct order — that is, the FIFO order in the absence of replication. AMPIPE automatically distributes and collects data to ensure this correct ordering. For instance, in mapping shown in Figure 4.1d, it automatically ensures that half the data goes to one copy and the other half to the other copy of module C. In addition, it also puts the data back together in the right order so that module D sees data in the same order as it would have if C hadn't been replicated.

4.2 Design and Implementation

In this section we present a design and implementation of AMPIPE, a streaming-pipeline application development system that is easy to use and doesn't require the users to deal with mapping, communication and synchronization. In Figure 4.2 we show the overall structure of AMPIPE

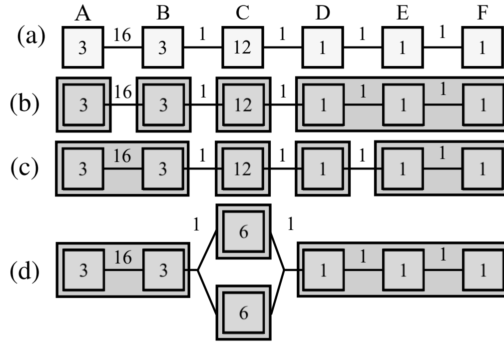


Figure 4.1: A pipeline example with various mappings.

which comprises of three separate parts : 1) a ***front-end*** which parses the user's pipeline, and allows the user to profile the pipeline, and generates the code for the pipeline (given a mapping); (2) a ***mapping*** pass that calculates the good mapping and this mapping is used in code generation; and (3) a ***runtime system*** which schedules and executes the application using the mapping.

4.2.1 AMPIPE front-end

AMPIPE's front-end is an extension of Scalapipe. It provides the programming interface, the profiler and code generation.

Programming interface The aim of AMPIPE's programming interface is to provide users the ability to easily create streaming pipeline applications at a high level. We adopt most of the programming interface from SCALAPIPE [74]. However, unlike SCALAPIPE, users do not have to provide a mapping and do not have to indicate which edge implementation to use for communication between modules; AMPIPE automatically calculates these. In order to calculate these, AMPIPE does require some information about each kernel, namely its runtime and its memory footprint each time it fires.

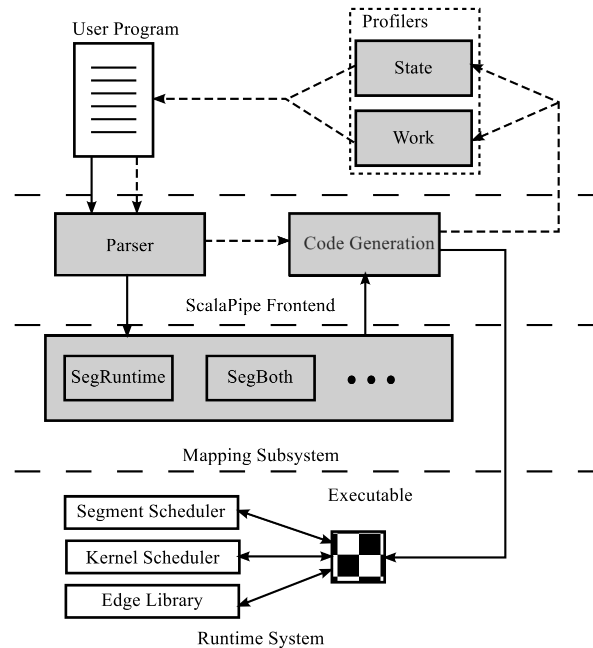


Figure 4.2: Architecture of the AMPIPE platform.

In Figure 4.3 we show a full "Hello, world!" application. The users responsibilities are to 1) define each individual computation kernel 2) describe the pipeline structure by chaining kernels together and 3) provide information about the available computing resources. This last piece allows AMPIPE to compute a good mapping and includes the number of cores available and the size of the highest-level private cache (Lines 28 and 29). The cache size is only needed if the user wants to use a mapping algorithm that takes cache misses into consideration.

Each kernel's definition includes the data type and size of their input and output as well as the number of data items read and written each time the kernel fires. (Source and sink kernels are exceptions since they lack output and input edges respectively.) The pipeline in Figure 4.3 contains two kernel definitions `BarKernel` and `BazKernel` respectively. Since `BarKernel` is a source kernel, it only provides the output edge, and indicates that it writes one item of type `STR` to its output edge each time it fires (Lines 6 and 7). Once declared, the output is simply used as a variable within the kernel; sending data along the edge is accomplished by assigning a value to

the variable (Line 9). In addition, it indicates that its running time is 1ns and that its state size is 64 Bytes (Lines 4 and 5). Finally, the kernel specifies its computation which can be arbitrary, but in this case it must contain one instruction that says `out = ..` to ensure that some string gets written to the output edge. Similarly `BazKernel` is a sink kernel; Lines 17 and 18 show its computation and memory requirements, Lines 19 and 20 shows how it reads from its incoming edge, and Lines 21—23 are its computation.

The actual pipeline is created in Lines 31 and 32 which indicate that `BarKernel` is the first kernel and sends its input into `BazKernel`. Line 34 initializes the pipeline itself. Users may also utilize external functions within their kernel allowing users to take advantage of available C/C++ programming libraries. A simple example is shown in Lines 11—15. This functionality is must more flexible and powerful than shown in this example and dramatically simplifies the development process; in fact most of the applications used in this paper take advantage of external C++ functions.

Profiling the Pipeline As mentioned above, to do automatic mapping, AMPIPE needs information about each kernel's runtime and memory usage. If the user does not already have the necessary profiling information, they can use AMPIPE's provided utilities to collect it by running a profiling pass which runs the pipeline and produces runtimes and memory footprint for each kernel and saves intermediate object files. AMPIPE includes two python utilities that will read this saved information and display it in a readable format so that the user may plug it into their kernel definitions.

Code Generation The final component of AMPIPE's front end involves generating the code for the user's application. AMPIPE adopts utilities from SCALAPIPE for producing the code for each individual kernel. AMPIPE has two main additions to the code generated: 1) the edge library,

which the different kinds of communication channels that may be used, and 2) the underlying runtime system which handles the scheduling and execution of the pipeline. The details for these additions are presented in Section 4.2.3.

4.2.2 Mapping and Code Generation

Mapping and scheduling algorithms are an important component of AMPIPE. AMPIPE provides a few default mappers that are very good at providing good load balance. The automatic mapping of pipelines allows users to quickly develop and deploy applications by removing the task of making assignment decisions from their workload. In addition, AMPIPE mapping framework utilizes a simple interface allowing users to easily define their own mapping algorithm.

AMPIPE's mapping framework is based on the idea of ***segmentation***. A ***segment*** is a set of contiguous modules restricted to the same processor. Note that a segment may contain anything between one module and all the modules — therefore, this policy can be quite general. How segments are chosen varies by scheduling policy. Grouping kernels in this way allows us to create a distinction between the expensive synchronized ***cross edges*** connecting neighboring segments and the cheap ***internal edges*** connecting kernels within a segment. AMPIPE automatically calculates the buffer that must be allocated to internal edges using the neighboring kernels input and output rates; it basically calculates the smallest buffer size necessary to prevent deadlock. The mappers decide how to size buffers for cross edges. Here we first briefly describe two mapping default mapping strategies provided by AMPIPE (previously discussed in Section 2.2.3). We then discuss how AMPIPE provides replication of modules. Finally, we explain briefly how users can write their own mappers in AMPIPE.

SEGRUNTIME: Evenly distributing work In many compute bound applications it is sufficient to find a mapping of kernels to cores that minimizes the total work assigned to any one computing resource. In particular, SEGRUNTIME finds exactly P segments such that the maximum load of any segment is minimized over all such segmentations. It uses dynamic programming to calculate such segments. Note that we can not simply use the computation time given by the user directly. Since each kernel may have different input and output rates, different kernels may fire different number of times over the course of the entire computation. Therefore, we first compute the ***normalized load*** and use this in the dynamic program. It is quite easy to see that using this strategy is within a factor of 2 of optimal assignment if runtime is the only factor. In addition, SEGRUNTIME makes the cross edge buffer large enough constant (in our case, it is 100 times the size of the data on the edge) that modules don't block due to lack of data on the cross edges.

SEGBOTH: Minimizing total load In Chapter 2 we show that for computations which are not compute-bound – those which employ a large amount data movement – relying solely on kernel workload may not be sufficient for finding mapping with good throughput. In particular, how we map an application has an impact on how many cache misses the application incurs and if we only consider runtime, then we can massively increase the number of cache misses incurred by the application, causing a reduction in throughput. To see this, consider again the example application shown in Figure 4.1. If we only consider runtime, then we may put modules A and B in different segments and therefore they will execute on different processors. This means that the edge from A to B is a cross edge and all the data going from A to B has to be written out of provate caches and read back in. Since the edge from A to B carries a lot of data, this causes many cache misses. On the other hand, if we keep A and B in the same segment and carefully schedule the segment then the data going from A to B can remain within the private cache of this processor avoiding these cache misses. Therefore, intuitively, we want to avoid (if possible) segment boundaries (cross edges) at expensive high-volume edges. SEGBOTH, like SEGRUNTIME, also tries to create segments

while minimizing the maximum load across all segments; however, it considers both runtime and cache misses when calculating the load. In particular, it first profiles the machine to calculate the cost of each cache miss. Then, at a high-level, for each possible segment, it calculates the total load by summing the runtime of the segment with the time it takes due to cache misses when it reads data from incoming cross edge and writes to the outgoing cross edge. It then uses dynamic programming to find the best possible segmentation. It does so by guessing at the load, and then trying to find a segmentation with that or smaller load using dynamic programming. It wraps this in a binary search for the best load. In addition, it allocates size M buffers on each cross edge (where M is the size of the last level cache as indicated by the programmer or by profiling).

Exploiting data-parallelism Distributing kernels to different cores exploits the explicit task-parallelism described by the pipeline. In many cases, applications do not have sufficient task-parallelism to fully utilize the provided computing resources. For example, consider one of our benchmarks, LZ77, whose pipeline only has 3 kernels. Without replicating one or more of the kernels the application would not be able to take advantage of the available resources. Alternatively, it may be that one kernel makes up over a majority of the computation. In this case, rather than distributing kernels evenly across cores, the bottleneck kernel should instead be replicated enough to allow for an even distribution of load.

At a high-level, the replication mappers find a minimal load mapping in two stages. It first computes the minimum and maximum load of all possible groups of contiguous kernels, where the first and last kernel of each group may be replicated. Next it finds the smallest total load any computing resource can be assigned, which yields a valid mapping, via binary search and dynamic programming. Why do we only consider replicating the first and the last kernel of any segment? Note that we only want to replicate bottleneck kernels. If a kernel in the middle of a segment is a

bottleneck, then we would just split this segment into multiple segments. The replication algorithm also decides what fraction of the data goes to each of the replicated modules.

AMPIPE supports replication for both SEGRUNTIME and SEGBOTH; the sole difference between the two provided replication mappers is the specific load calculation. SEGRUNTIME with replication computes load as a function of work, whereas SEGRUNTIME with replication includes both the work and an approximation of the cost of the memory accesses required from performing input and output.

Replication in AMPIPE is done *as needed*. If a mapping of the pipeline can be found that evenly distributes the total load across cores without the use of replication there is no benefit to replicating kernels. Indeed replication is only beneficial when there exist bottleneck kernels in the pipeline or when there are not enough kernels assign to all available resources.

An extensible mapping framework At a high-level, the responsibilities of the mappers in AMPIPE are to 1) assign kernels to cores in an informed manner and 2) to determine the buffer sizes on cross edges between each kernel. To write their own mapper, the user first implements a partitioning function based on the kernel information provided by the user to create all the segments. It then decides how to assign segments to cores (more than one segment can be assigned to a single core). Finally, it must decide how to allocate buffers on cross edges by using either one of the default strategies or by defining their own. Mappers are hooked into the front-end via a single map routine which processes the pipeline and performs the three key operations.

4.2.3 Runtime

AMPIPE's runtime has a few different functions: First, it must schedule each segment on one processor and decide when each segment executes. Second, it must decide which kernel within each segment to execute at any time. Finally, it must manage communication between kernels via edges in the pipeline. Alongside the code generated for the user's kernels, AMPIPE produces a library which includes the necessary scheduling routines for threads, segments and kernels as well as various communication channels to support the different kinds of edges that the mapping may generate.

Thread scheduling The thread scheduling in AMPIPE is straightforward. Each segment is assigned to a single thread and each thread is pinned to a single processor. However multiple segments may be assigned to the same thread. The resulting application created by AMPIPE is a multi-threaded process where each thread manages it's set of assigned segments individually. When assigned multiple segments, AMPIPE's threads prioritize executing those which are found the farthest downstream and are *ready* for execution. A segment is *ready* for execution when the incoming cross edge into the segment is at least "half full" and the outgoing cross edge is at least "half empty." Recall that AMPIPE assigns large buffers to cross edges and this half full/half empty criterion makes sure that once a segment is scheduled, it runs for a while before another segment needs to be scheduled. This reduces overheads of warming up the cache, etc. If multiple segments on the thread are ready, AMPIPE schedules the one that is farthest downstream since this allows us to finish emptying the pipeline faster. Once a segment is fired, it continues executing until either its incoming cross edge buffer is empty or its outgoing cross edge buffer is full.

Segment scheduling Segments are lists of consecutive kernels which are fired as a single unit. A kernel is ***ready*** when there is enough data in its input edge buffer to satisfy its input requirements

and enough space on its output edge buffer to satisfy its output requirements. Within a segment, we use the same strategy as we use across segments — that is, we always fire the furthest downstream kernel that is ready to fire within a segment. The buffers on edges within the segment are large enough to guarantee that we can always find some ready kernel in a segment as long as the segment itself is fireable. When one of a segment's kernels is replicated, the segment scheduling may be slightly more complicated since instead of just one incoming (and outgoing) edge, the segment may have more than one incoming (or outgoing edge). In this case, the segment keeps internal counters to keep track of the elements on each edge and appropriately decides which kernel to fire.

Communication Library One of the main goals of AMPIPE is to remove the challenge of handling communication from the user; to that end we include a number of different edge classes, each designed for a different style of communication while using the same interface. When designing their computation kernels, the user need not worry about which type of edge is used to communicate — they simply use the interface described earlier. The type of communication necessary for any given edge is determined not by the user, but during the mapping stage of AMPIPE. The runtime will automatically use the correct type of edge depending on the type of communication necessary. AMPIPE includes 5 communication types.

Internal Edges: Simple non-synchronized edges which are used between kernels within the same segment are implemented as simple buffers since the same thread is both the producer and consumer.

Cross Edges: Thread-safe, single-producer single-consumer edges which can be accessed by at most two threads to communicate between two segments when neither of the two end-points of the edge are replicated modules. A simple thread safe ring buffer is used in this case. This keeps the

buffer from overflowing and also prevents readers from reading from an empty queue by synchronizing on the number of items in the queue. Figure 4.7 shows the read and write methods of our thread safe queues.

Split Edges: Thread-safe, single producer multiple-consumer edges which splits data amongst multiple threads. These edges are used during replication when the consumer kernel exists in more than one segment and the producer only exists on one. For instance, in Figure 4.1, in the replicated mapping, this edge connects module B to multiple copies of module C. The mapping algorithm decides which fraction of the data must go to each of the replicated copies of a module and split edge implementation is responsible for enforcing these decisions. In our current implementation, split edges first sends its entire portion of the data to the first copy of the module, then to the second copy and so on. The alternative would be distribute data in a round-robin manner, but we find that this contiguous partitioning works better.

Join Edges: These are counterpart to split edges and are thread-safe, multiple-producer single-consumer edges which collect data from multiple threads. These edges used during replication when the producer kernel exists in more than one segment. These edges must make sure that data ordering is preserved — that is, to the consumer kernel, it should look like the preceding kernel was not replicated and executed the elements in the correct order. Therefore, AMPIPE edge implementation carefully reads data from multiple producers in the correct order. We've designed as our split and joine edges as collections of thread-safe queues. Where only the producer or consumer interacts with the split or join edge interfaces respectively. Figure 4.8 shows our Split/Join edge implementations. Each edge keeps a set of TSPQ edges which the replicated kernels will interact with and keeps track of when the edge must switch amongst them.

Interchange Edges: Thread-safe multiple-producer multiple-consumer edges which bridge communication between two neighboring kernels which have both been replicated. An Interchange

edge connects multiple ITSPQ edges which serve as the endpoints to each of the replicated procuders and consumers. Given the order of the incoming producer and consumer ITSPQ edges and their sizes, the Interchange determines the start and end pointers for each ITSPQ within its own buffer. With these endpoints, the Interchange breaks the buffer into regions which can be assigned to ITSPQs such that for each region there is exactly one producer and exactly one consumer, which an ITSPQ may be assigned multiple neighboring regions. To prevent races, ITSPQs grab a lock on region whenever performing a read or write. Figure 4.9 shows this implementation.

From the perspective of a kernel, the actual type of edge does not matter, it utilizes the same interface to pull or push data from any of them. Each edge type also preserves the original ordering of data, a task especially important in the context of replication.

4.3 Experiments

In this section we explore how AMPIPE handles a number of real-world streaming-pipeline applications. To demonstrate the features included in AMPIPE we implemented DES, LZ77, Ferret and a deep neural-network (DNN) testing application. We used AMPIPE interface to program these applications and ran then using SEGRUNTIME and SEGBOTH with and without replication. We find in all cases that AMPIPE's automatic mapping produces applications who's performance is able to scale with increasing computational resources. In particular, we find that when we have pipelines which are compute bound or which have similar data volume across all edges, SEGRUNTIME and SEGBOTH find the same mapping and perform comparably. When these are not true (in DNN), SEGBOTH performs better. In addition, replication is really useful in getting good performance for pipelines with bottlenecks (ferret and LZ77).

App-Mapp	1	2	4	8
des-sr	404.3 (1.0)	230.4 (1.8)	142.0 (2.8)	96.5 (4.2)
des-sb	404.4 (1.0)	213.5 (1.9)	127.8 (3.2)	78.2 (5.2)
lz77-sr	148.6 (1.0)	150.9 (1.0)	154.4 (1.0)	154.3 (1.0)
lz77-sb	148.6 (1.0)	153.2 (1.0)	154.9 (1.0)	154.4 (1.0)
lz77-srr	148.6 (1.0)	113.6 (1.3)	54.3 (2.7)	34.1 (4.4)
lz77-sbr	148.6 (1.0)	115.7 (1.3)	54.9 (2.7)	34.9 (4.3)
ferret-sr	550.1 (1.0)	521.4 (1.1)	367.2 (1.5)	366.3 (1.5)
ferret-sb	562.0 (1.0)	379.3 (1.5)	382.0 (1.4)	379.3 (1.5)
ferret-srr	550.1 (1.0)	333.2 (1.7)	184.8 (3.0)	118.7 (4.6)
ferret-sbr	562.0 (1.0)	326.9 (1.7)	179.1 (3.1)	113.5 (4.8)
dnn-sr	269.6 (1.0)	144.0 (1.9)	84.4 (3.2)	51.9 (5.2)
dnn-sb	269.6 (1.0)	139.2 (1.9)	71.2 (3.8)	37.6 (7.2)

Table 4.1: Runtimes (with speedup over serial in parenthesis) for DES, LZ77, Ferret, and DNN. sr is SEGRUNTIME without replication, srr is SEGRUNTIME with replication, sb is SEGBOTH without replication and sbr is SEGBOTH with replication. For DES and DNN, mappers with and without replication find the same mapping.

Experimental Setup All experiments were run on an 8-core Intel Xeon E5-2687W with a 20MB SmartCache, running at 3.1GHz with hyperthreading enabled. Though the streaming model assumes an infitite input stream, most real streaming applicaitons work on a finite dataset. We selected inputs large enough to amortize the startup and cleanup phases of the pipeline execution.

DES **D**ata **E**ncryption **S**tandard was a federal encryption standard in the United States until 2002 when it was superseded by the Advanced Encryption Standard. Figure 4.10a shows the DES pipeline consisting of a series of Feistel function and permutation kernels that process 64 bit blocks of data. Since the data volume of every edge is the same (each kernel consumes and produces 64 bits of text on each firing) both SEGBOTH and SEGRUNTIME produce the same mapping, dividing up the kernels evenly into 8 segments; since this pipeline has enough kernels, no replication is necessary.

Although the segmentations found by both SEGBOTH and SEGRUNTIME are the same, Table 4.1 shows that SEGBOTH slightly outperforms SEGRUNTIME on DES. This difference is due to the buffer allocation strategies used by each mapper. SEGBOTH, in this case, incidentally creates large buffers (the size of cache). This allows each segment to execute for longer when it is scheduled, thereby reducing idle time.

LZ77 LZ77 is a lossless, dictionary file compression algorithm which maintains a sliding window view of the input and removes duplications by replacing them with forward references. Like DES, LZ77 streams input files through a series of computations and thus fits naturally into the streaming model. Figure 4.10b shows the LZ77 pipeline which is made up of Reader and Writer kernels for performing file i/o and a central compress/decompress kernel which is responsible for the majority of the work.

With such few kernels, LZ77 does not have enough task parallelism. The original LZ77 algorithm can not directly use replication in the middle kernel since the sliding window requires the order of data to remain unchanged. Instead, we implemented a variation of LZ77 which resets the window at set intervals, removing dependencies and exposing data-parallelism. This variation splits the input into blocks of 16384 characters which can be compressed individually, allowing us to mark the compression kernel as replicable.

Again, SEGRUNTIMER and SEGBOTHR create the same mapping, shown in Figure 4.10c. For P cores, both mappers create P copies of the Compress kernel and assigns one to each core. The Reader and Writer kernels are both paired with one copy of the Compress kernel each. Since Compress operates on larges blocks of data at a time, the difference between the buffers allocated SEGRUNTIME and SEGBOTH is minimal. Therefore, the difference between the running time of the two mappers is minimal. In contrast, if we do not use replication, LZ77 slows down as we increase the number of cores. This is due to the fact that most of the work is still done by one

module and we have simply added the overheads of data transfer between modules. Therefore, replication is essential to get performance for this pipeline.

Ferret Ferret, which has been included in the parsec benchmark suite using a pipeline implementation, is a content based similarity search application. We show the parsec pipeline in Figure 4.10d which is broken up into 5 compute kernels. Profiling the pipeline reveals an uneven distribution of work amongst the kernels. As we can see, the Rank kernel is the bottleneck. Figure 4.10e shows the ferret pipeline after replication on 8 cores. As we can see, it is not always sufficient to just replicate the largest bottleneck. After creating enough replications of the Rank kernel, vec kernel emerges as the bottleneck. Rather than blindly replicate Rank accross all cores, our mappers automatically identify when a new kernel has become the bottleneck and replicate it in order to continue to reduce the maximum load. Again, SEGBOTH and SEGRUNTIME find the same mapping and provide the same performance. However, the mapper without replication provides very little speedup due to the bottleneck module, while replication provides considerable performance advantage and scalability as we increase the number of cores.

DNN We've implemented a neural-net testing computation which we call DNN. For this experiment, our goal is not to produce a meaningful interpretation of the input data, but rather to demonstrate how a neural network might be tested as a parallel streaming application. Our DNN pipeline is made up of a series of hidden layers which we combine into individual kernels. Layers are made up of a collection of perceptrons which check if the weighted sum of it's inputs is greater than it's assigned threshold. Perceptrons are connected to the previous layer, observing the outputs of a fixed set of perceptrons from that layer. For our application, each layer has the same number of perceptrons and is characterized by how connected it is to the previous layer. The DNN pipeline can be found in Figure 4.10f. It is made up of 64 hidden layers which have varying connectivities.

P	SEGRUNTIME	SEGBOTH
1	237566435	237566435
2	164066083	136108168
4	175842671	83401537
8	232859976	42227065

Table 4.2: L2 Cache miss counts for DNN with for SEGRUNTIME and SEGBOTH

We created a network where 1/4 of the edges are low volume edges between layers with low connectivity, while the remaining edges all have high volume. DNN is a memory bound application and the data volume across different edges varies — therefore, as can be expected, SEGRUNTIME and SEGBOTH find different segmentations.

Figure 4.10fa shows the segmentation found by SEGRUNTIME, while Figure 4.10fb shows the segmentation found by SEGBOTH. SEGRUNTIME is strictly concerned with distributing the work of individual kernels evenly causing it to select high-volume edges as cross edges. On the other hand, SEGBOTH avoids this and picks low-volume edges as cross edges. In Table 4.1 we see that the mapping found by SEGBOTH does indeed outperform SEGRUNTIME and gets a speedup of 7.2 on 8 cores. A closer inspection of the cache misses in Table 4.2 indicates that this reduction in runtime is due to a significant reduction in the number of cache misses.

4.4 Related Work

How to mapping of pipelines has been studied extensively, mostly without replication for both homogenous and heterogenous platforms and here we do not review this work and just provide a few citations [4, 15, 17, 29, 38, 59, 62].

Pipeline Scheduling Systems A number of pipeline scheduling systems already exist which aim to provide a high-level pipeline programming interface with efficient runtime scheduling. The closest related works are the X language, AutoPipe and ScalaPipe [32, 74] which all serve as ancestors of AMPIPE. These were both designed for easily creating pipelines for heterogenous platforms. Users develop kernels in C or VHDL (in X) and in Scala (in ScalaPipe) and then can run them on CPUs, FPGAs, GPUs, etc. In these systems mapping is always the responsibility of the user. Should a set of kernels be mapped to a CPU resource, scheduling is managed dynamically by the OS rather than directly within a runtime as in AMPIPE. Similarly, a recent work on RafLib [11], a C++ template library that aims to enable high-performance stream processing while utilizing a natural C++ interface that provides a minimal learning-curve, also does not provide automatic mapping, especially if we consider cache misses.

StreaMIT [71] offers a stream-based programming language that supports general streaming applications beyond pipelines. The StreaMIT compiler uses a phased scheduling strategy to minimize latency, as oppose to minimizing throughput, which is the focus of AMPIPE. Gordon et al. [36] extends this work to multiple cores by evenly distributing load using a greedy assignment while only considering work and not cache misses for replication. Lee et al. [47] explore beyond these construct-and-run systems and proposed the piper algorithm which tackles on-the-fly pipeline parallelism. Cilk-P, their prototype implemetation, discovers the pipeline structure at during execution and utilizes the Cilk runtime system to dynamically schedule and balance load accross multiple cores.

Replication Replication has been considered in some limited cases. Subhlok et al. considered a model where every task can be perfectly parallelized [68] as well as solutions for latency-throughput trade-offs [69]. Kudlur et al. and Cordes et al. used integer linear programming to extract data parallelism from streaming pipelines [30, 44]. While they considered replication, they

all assumed that the work of replicated stages should be *evenly* divided and assigned to replicas. Li et al. [**?**] provided an algorithm for replication on heterogenous platforms and with uneven division. Wang et al. proposed a machine learning-based approach to mapping streaming applications on multi-cores [73]. However, all these replication approaches only consider runtime and do not consider cache misses. AMPIPE is unique in being able to add replication even when we consider cache misses to the load balance criterion.

4.5 Conclusions

In this chapter we have presented AMPIPE, an extensible, easy-to-use parallel streaming-pipeline application development system that finds good static mappings automatically using profiling information. The application generated is a multi-threaded C++ program built ontop of AMPIPE's underlying runtime system which utilizes a communication library to facilitate data movement and synchronization. Experimental results show with automatic replication and mapping AMPIPE can continue to achieve speedup on real-world applications as you increase the available computing resources.

```
1   object FOO extends App {
2       val BarKernel = new Kernel("Bar") {
3           /* Kernel setup */
4           config(state, 64)
5           config(runtime, 1)
6           config(outrate,1)
7           val out = output(STR)
8           /* What to execute */
9           out = "Hello␣World!"
10      }
11      /* An external C function to be used within a kernel */
12      val printf = new Func("printf") {
13          include("stdio.h")
14          external("C")
15      }
16      val BazKernel = new Kernel("Baz") {
17          config(state,64)
18          config(runtime,1)
19          config(inrate,1)
20          val in = input(STR)
21          val instr = local(STR)
22          instr = in
23          printf("%s\n",instr)
24      }
25      /* The application */
26      val app = new Application {
27          /* Application paramters and system info*/
28          param(cache,262144)
29          param(cores,2)
30          /* pipline creation */
31          val bar = BarKernel()
32          BazKernel(bar)
33      }
34      app.emit("foo")
35  }
```

Figure 4.3: A full “Hello, World” pipeline programming demonstrating kernel,pipeline and application definitions including the newly required calls to pass profiling and system information to the front-end.

```
1   /* Abstract Mapper class definition */
2   private[scalapipe] abstract class Mapper(val p: pipe) {
3       /* Compute the edge gains*/
4       def compute_gains()
5       /* Assign minimum buffers for all edges*/
6       def assign_min_buffers)()
7       /* Abstract methods */
8       def create_segments(): Unit
9       def assign_segments_to_cores(): Unit
10      def map() {
11          compute_gains()
12          assign_min_buffers()
13          create_segments()
14          assign_segments_to_cores()
15      }
16  }
```

Figure 4.4: AMPIPE's abstract Mapper class which provides the interface for defining new Mappers which at a minimum requires users to implement the create_segments and assign_segments_to_cores methods.

```
1   bool schedule_maxload(long load,segments * segs);
2   /* The SegBoth algorithm*/
3   void create_segments() {
4       compute_all_loads()
5       /* Find load budget */
6       L_min = L_max = −1;
7       L_guess = 1;
8       segments * segs;
9       while(L_max == −1) {
10          if (!schedule_maxload(L_guess,segs))
11              L_min = L_guess ; L_guess *= 2;
12          else
13              L_max = L_guess;
14      }
15      /* Search between L_min and L_max */
16      done = false
17      L_next = mid(L_min,L_max) /*midpoint*/
18      while(!done){
19          L_guess = L_next;
20          if (!schedule_maxload(L_guess,segs))
21              L_min = L_guess ;
22          else
23              L_max = L_guess
24          L_next = mid(L_min,L_max)
25          if (L_guess == L_next)
26              done = true
27      }
28      /* segs now holds the segmentation*/
29  }
```

Figure 4.5: The SEGBOTH segmentation algorithm. schedule_maxload is a helper function which determines if a segmentation can be found within a given load and if so, stores it.

```
1   /* Determine start and end kernels*/
2   int start,end;
3   int fireKernelNum = start
4   for(;;) {
5       if (kernelList[fireKernelNum]->firable()) {
6           kernelList[fireKernelNum]->run();
7           update_read_count();
8           update_write_count();
9           if (fireKernelNum != end)
10              fireKernelNum++;
11      }
12      else {
13          if (fireKernelNum == start+1)
14              break;
15          else
16              fireKernelNum--;
17      }
18  }
```

Figure 4.6: Kernel scheduling function within a segment. Each firing results in the first kernel of the segment executing once before propataing it's output as far as possible through the segment.

```
1   template <typename T>
2   void TSPQ<T>::read(void * l) {
3       /* Cast the templated type */
4       T* loc (T*)l;
5       T* buff = (T*)this->m_buff;

7       /* Spin while the buffer is empty*/
8       while(empty())
9           ;
10      auto curr_pos = m_read_pos.load();
11      memcpy(loc,&buff[curr_pos],sizeof(T));
12      m_read_pos.store((curr_pos+1) %this->m_size,std::memory_order_release);
13  }

15  template <typename T>
16  bool TSPQ<T>::empty() {
17      /* Check if the read and write pointers are the same */
18      return m_write_pos.load(std::memory_order_acquire) == m_read_pos.load(std::
            memory_order_acquire);
19  }

21  template <typename T>
22  void TSPQ<T>::write(void * v) {
23      /* Cast the templated type */
24      T* val = (T*)v;
25      T* buff = (T*)this->m_buff;

27      /* Spin while the buffer is full */
28      while(full())
29          ;
30      auto curr_pos = m_write_pos.load();
31      memcpy(&buff[curr_pos],val,sizeof(T));
32      m_write_pos.store((curr_pos+1)%this->m_size,std::memory_order_release);
33  }

35  template <typename T>
36  void TSPQ<T>::full() {
37      /* Check that the write pointer is not right behind the read pointer */
38      auto curr_pos = m_write_pos.load(std::memory_order_acquire);
39      return (curr_pos+1)%this->m_size == m_read_pos.load(std::memory_order_acquire);
40  }
```

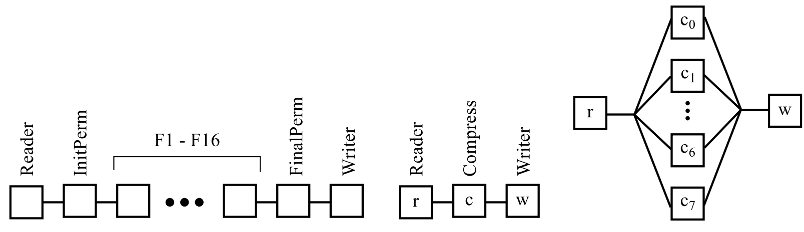
Figure 4.7: AMPIPE's single producer single consumer thread safe ring buffer used for cross edges between non-replicated modules.

```
1   /* Split edge write method */
2   void STSPQ::write(void * val) {
3       /* Get the queue */
4       Edge * queue = m_queues[m_write_pos];

6       while(queue->full())
7           ;
8       /* Call the underlying queue's write */
9       m_queues[m_write_pos]->write(val);
10      m_cnt++;
11      /* Check if we need to change queues */
12      if (m_cnt == m_counts[m_write_pos]) {
13          m_write_pos = m_write_pos < m_splits-1 ? m_write_pos+1 : 0;
14          m_cnt = 0;
15      }
16  }

18  /* Join edge read method */
19  void JTSPQ::read(void * loc)
20  {
21      /* get the queue */
22      Edge * queue = m_queues[m_read_pos];
23      while(queue->empty())
24          ;
25      /* Call the underlying queue's read */
26      m_queues[m_read_pos]->read(loc);
27      m_cnt++;
28      / * Chekc if we need to change queues */
29      if(m_cnt == m_counts[m_read_pos]) {
30          m_read_pos = m_read_pos < m_splits-1 ? m_read_pos+1 : 0;
31          m_cnt = 0;
32      }
33  }
```

Figure 4.8: AMPIPE's Split and Join edges, which are responsible for trafficking data between replicated and non-replicated kernels. Only one producer will be connected to a split edge, while only one consumer will be connected to a join edge, making the read and write methods unnecessary in those two classes respectively.

```
1  template <typename T>
2  Interchange<T>::Interchange(uint64_t _size, int _writers, uint64_t *_wlens,
3                              int _readers, uint64_t *_rlens) :
4      size(_size),writers(_writers),write_lens(_wlens),
5          readers(_readers),read_lens(_rlens) {
6      find_regions();

8      /* create per-region flags / locks */
9      full_flags = new bool[nregions];
10     counts = new uint64_t[nregions];
11     muts = new std::mutex[nregions];

13     /* Assign regions to the ITSPQs */
14     assign_regions();
15 }

17 template <typename T>
18 void ISPQ<T>::read(void * l)
19 {
20     T* loc = (T*) l;
21     T* buff = (T*) it->buff;

23     while(empty())
24         ;

26     /* Get this reader's region*/
27     int curr_pos = (*it).read_pos[id];
28     uint64_t region = it->read_regions[id*it->nregions+region_pos];

30     {
31     /* lock the region */
32     std::lock_guard<std::mutex> lk(it->muts[region]);

34     memcpy(loc,&buff[curr_pos],sizeof(T));

36     it->counts[region]--;

38     /* Check if we've finished a region and change if so */
39     if (curr_pos == it->regions[region]-1)
40     {
41         it->full_flags[region] = false;
42         int end = it->read_regions[id*it->nregions];
43         region_pos = (region_pos == end) ? 1 : region_pos + 1;
44     }
45     } /* lock released */

47     /* Update the position read position */
48     if (id == 0)
49         it->read_pos[id] = (curr_pos == it->read_ends[id]-1) ? 0 : curr_pos+1;
50     else
51         it->read_pos[id] = (curr_pos == it->read_ends[id]-1) ? it->read_ends[id-1] : curr_pos + 1;
52 }
```

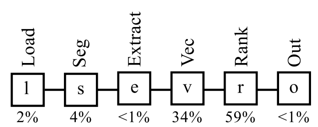
Figure 4.9: The Interchange and ITSPQ classes responsible for trafficking data between two replicated kernels



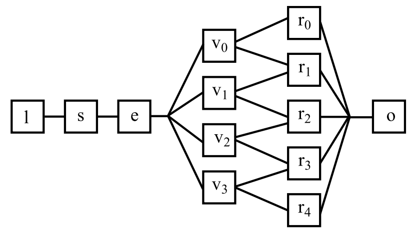
(a) The pipeline configuration for DES. It has two initialization and two finalization kernels and in the middle there are 16 functional kernels.

(b) The pipeline for LZ77; it contains only 3 kernels, therefore, it is essential that we take advantage of data parallelism using replication.

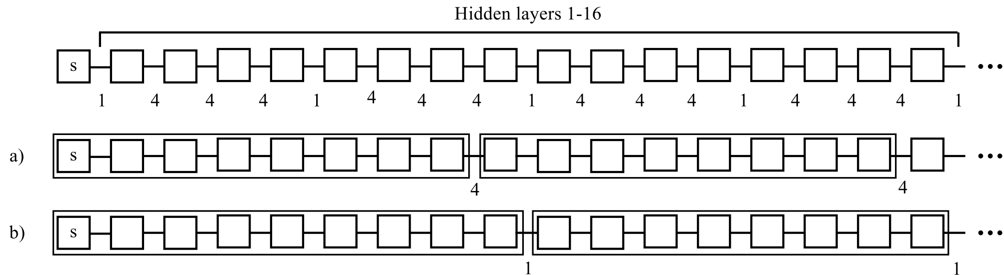
(c) LZ77 pipeline after replication for both SEGRUNTIME and SEGBOTH. At 8 cores, $\{r, c_0\}$ are assigned together, $\{c_7, w\}$ are assigned together and the remaining c copies are assigned alone.



(d) Pipeline configuration for ferret. This application has unbalanced loads and the rank module is initially a bottleneck. However, once that module is replicated, Vec module becomes a bottleneck.



(e) Ferret pipeline after replication. Two modules are replicated — therefore, this application uses all kinds of edges — split edges between e and v, interchange between v and r and join between r and o. The mappers put together $\{l, s, c, v_0\}$, $\{v_3, v4\}$, $\{v_0, 0\}$,and assign the remaining v and r copies alone.



(f) Pipeline configuration for DNN, we show only the first 17 kernels for space. This applicaiton has many more kernels than cores providing plenty of options for how a mapper might divide them up.

Chapter 5

Locality-Aware Dynamic Task-Graph Scheduling

5.1 Intro

In recent years a number of high-level programming languages and libraries have emerged, such as OpenMP, Cilk Plus, Nabbit, etc. These languages and libraries allow programmers to express the logical parallelism in their programs while the runtime scheduler schedules the work on the available cores automatically. For multicores with few cores and uniform access to the memory hierarchy, these languages and runtime systems provide both good performance and a relatively simple programming model.

On large multicores with non-uniform memory access (NUMA), however, *locality* is an important consideration since a *remote memory access*—access to data reachable from a memory controller that is further away via the on-chip network—can cost much more than a *local memory access*. Regular applications can be structured to implicitly ensure locality between initialization and subsequent use when using static schedulers such as in OpenMP. However, irregular applications need dynamic load balancing which dynamic schedulers, such as those in OpenMP tasks, TBB, and Cilk

Plus provide. However, they have no notion of the location of data and often fail to provide good performance for regular memory-intensive applications.

Ideally, one would like to have a high-level and easy-to-use programming model which incorporates dynamic scheduling and locality. We present NABBITC, a locality-aware extension of a task graph library NABBIT. In the NABBIT programming model, the programmer expresses computations as a task graph where each node is a task and edges represent dependences between tasks. NABBIT is a library built on top of a Cilk Plus[3] and therefore, NABBIT programs are scheduled using a provably good work-stealing scheduler.

In this chapter, we make NABBIT locality aware by allowing the programmer to give locality hints to the scheduler using a simple coloring scheme. In particular, we make the following contributions.

1. We extend the interface so that the programmer can provide a *color* to each task; if a task is colored a color c, then the data used by this task is local to processor with color c. Multiple nearby cores can have the same color.

2. We modify both the NABBIT library and the Cilk Plus runtime system to allow processors to preferentially execute tasks that share the color with them. Therefore, if the user provides "correct coloring", then workers preferentially execute tasks that access local data, thereby reducing the expensive remote accesses.

3. NABBITC tries to strike a balance between improving locality and preserving the guarantees of low overhead and good load balance provided by NABBIT. We prove that NABBITC, by and large, preserves the asymptotic guarantees provided by NABBIT. In particular, for

[3] It was originally designed to be built on top of Cilk++, but it is trivial to port to Cilk Plus. Indeed, it was designed so that it can be ported to any programming language that supports fork-join parallelism.

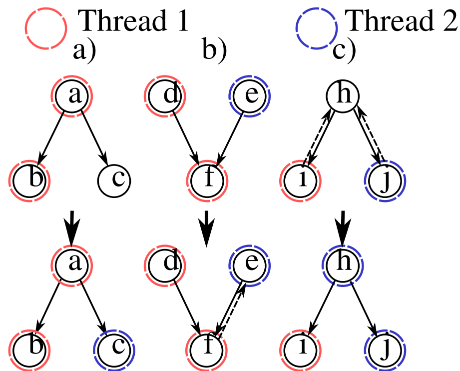


Figure 5.1: NABBIT scheduling examples presented in two stages. A thread surrounding a node denotes that thread is processing that node. A solid line from a to b denotes that b is a predecessor of a while a dashed line from b to a denotes that a is in b's successor list.

reasonable task graphs—those with enough parallelism and where tasks of all colors appear near the root of the graph—NABBITC provides nearly asymptotically optimal speedup.

4. We evaluated the performance of NABBITC on a suite of memory intensive applications and find that it succeeds in providing both good load balance and good locality. It consistently out performs vanilla NABBIT due to improvements in locality. In addition, on PageRank, an exemplar irregular benchmark, NABBITC outperforms OpenMP static and guided scheduling strategies by combining dynamic load balancing and locality awareness.

5.2 Background

In this section, we describe NABBIT, a high-level task-graph scheduling library built on top of the Cilk Plus runtime system. We outline the NABBIT programming model and show how NABBIT recursively executes task graphs in parallel. We also provide a brief overview of the GCC Cilk Plus implementation upon which NABBIT is built.

NABBIT task-graph scheduling. NABBIT schedules task graphs through static and dynamic exploration of the task graph. A NABBIT task graph is a directed acyclic graph with a set of explicit nodes that represent tasks and edges represent dependences between tasks. Each node u in the task graph specifies its predecessors—tasks that have edges to u and therefore must be executed before u can be processed. For the paper, we will use the terms node and task interchangeably.

We summarize the key aspects of the NABBIT dynamic task graph scheduler (more details in Agrawal et al. [7]). One interesting property of NABBIT is that it computes nodes *on demand*. The scheduler takes an input specified in the form of a sink node, whose execution completes the execution of the task graph. Upon creation, this node has a list of predecessors and no successors. The sink node together with the predecessor specification transitively identifies all vertices that need to be executed to compute the sink node. The scheduler actions:

1. To process a node, a thread initializes the node and its list of predecessors and proceeds to execute them in a recursive parallel depth-first fashion. Consider the example in Figure 1a. When thread 1 wants to process a and finds that b and c are its predecessors that have not been initialized, it goes ahead and tries to process one (b in this case). While b is being processed, another thread can steal c.

2. When processing a node's predecessor, if a thread finds that some predecessor has already been initialized by some other thread but has not finished executing, the thread adds the current node to the predecessor's successor list and moves on. In Figure1b, thread 1 is processing d while thread 2 is processing e. f is a predecessor of both d and e. Each thread will try to initialize the predecessor f but only one will succeed, in this case thread 1. Thread 2 is attaches e to f's successor list and tries to find other work to do.

3. After a node is computed, the thread checks if there are any enqueued successors and if so, determines if those successors are ready to execute (i.e., have no other predecessors on which

they are waiting). In the event that a successor is ready, the thread will recursively execute that node. In Figure 1c, both nodes i and j have h enqueued in their successor lists. Thread 1 computes i and checks if h is ready to execute. Since h still depends on j, thread 1 moves on. Thread 2, after computing j checks h, sees that it is ready and proceeds to execute it.

This procedure ensures that a node is computed only after all its (transitive) predecessors have been computed, ensuring correctness. In addition until an initialized node u is computed, it is (a) either in a thread's stack, (b) in a successor list, or (c) is a predecessor of an initialized node. This guarantees that every node u will be inspected and executed eventually. Also, this ensures that the sink node, and thus whole task graph, is executed to completion.

Atomicity choices ensure the absence of data races. The predecessor and successor lists allow threads to execute without blocking/waiting for any action by another thread. The recursive parallel design allows for the implementation of the NABBIT's scheduler as a Cilk Plus program. All vertices in either the predecessor or successor lists can be executed in parallel. In addition, NABBIT ensures that no ordering constraints other than those implied by the predecessor relationships is imposed on the execution: a node u is ready to execute immediately after all its predecessors have been computed and unless every processor is busy doing other work, some processor will find and execute u. This ensures that NABBIT does not alter the task graph's critical path length, enabling the scheduler to guarantee asymptotic optimality. Essentially, if the task graph itself has a parallelism of at least P, then NABBIT guarantees that it gets $\Omega(P)$ speedup on P processors for most reasonable task graphs. In addition, since it leverages the Cilk Plus work-stealing scheduler and uses distributed processing, NABBIT provides low overheads. These properties of asymptotic optimality and low overheads are not normally achieved by other task graph schedulers, such as scheduler currently used in OpenMP's SMPSs [61], since they do not process nodes on demand.

GCC Cilk Plus We compile NABBIT using the GCC implementation of Cilk Plus, an extension to C++. Cilk Plus is a *processor oblivious* language—the programmer expresses the *logical parallelism* of the program using three keywords without any reference to how many threads must execute the program and how. The cilk_spawn keyword indicates that the succeeding function can execute in parallel with its continuation. The cilk_sync keyword is a local barrier; all previously spawned functions by current function must complete before the program execution can move this statement. Cilk Plus also provides a parallel_for keyword, which indicates that all iterations can be executes in parallel. This keyword is essentially syntactic sugar and is implemented using spawns and syncs.

The Cilk Plus runtime system uses randomized work stealing to schedule these fork-join programs on P available cores. The program executes on P worker threads, one for each core in the target machine. Each worker has a local deque of work. When a worker p executing function foo spawns a function bar, the frame corresponding to the caller foo is placed at the bottom of the p's deque and p starts executing bar. When p returns from a function, it pops the function at the bottom of its deque and continues executing. (If executing on one thread, the program follows the normal depth-first execution followed by C or C++.) If worker q's deque is empty, it becomes a thief, picks a random victim worker, say p, steals the top frame from p's deque, and starts executing it. If a steal attempt is unsuccessful, meaning that the victim had an empty deque, then the thief continues to steal until it finds work.The Cilk Plus compiler inserts code at spawns and syncs to ensure that deques are managed correctly. In addition, when a worker's deque is empty, it makes calls into the runtime to make sure that steals occur correctly.

5.3 Theory

We now present a simple analysis showing that the modifications made to NABBIT do not negatively effect the asymptotic runtime—this implies that NABBITC also provides almost asymptotically optimal load balancing for programs that have enough parallelism.

Just as in the NABBIT paper [7], say, we are given a task graph $G = (V, E)$, where each node u has work $W(u)$. Also, say that s is the unique node with zero in-degree and t is the unique node with zero out-degree. If these nodes are not unique, we can trivially add dummy root and final nodes. Define M as the number of nodes on the longest path in V from the source s to the sink t.

We can define the work T_1 as the time it takes to execute the task graph on a single processor and span T_∞ as the time it takes to execute it on an infinite number of processors. Therefore, the work is $T_1 = \sum_{u \in V} W(u) + O(|E|)$. The second term is due to the fact that each edge needs to be checked to make sure that it is satisfied. Similarly, we the span is $T_\infty = \max_{p \in paths(s,t)} \{\sum_{u \in p} W(u) + O(M)\}$ since nodes along any path through V can not execute in parallel. By the work and span laws [31, p. 780], the completion time on P processors for a task graph is at least $\max\{T_1/P, T_\infty\}$.

We will prove the following theorem—the analysis is a small extension to the analysis of runtime for NABBIT.

Theorem 5.3.1. *For task graph $G = (V, E)$ with maximum degree d,* NABBITC *executes G in time $O(T_1/P + T_\infty + M \lg d + \lg(P/\epsilon) + C)$ time on P processors with probability at least $1 - \epsilon$ where C is the amount of time each worker spends at startup trying to find a node of its own color.*

This theorem is similar to the theorem proved for NABBIT [7] apart from the last term C. The main difference between NABBIT and NABBITC is the fact of colored steals. In particular, when a worker runs out of work in NABBIT, it performs a random steal. On the other hand, when a worker runs out of work in NABBITC, it first checks a constant number of deques to see if it can find work

of its own color and only performs a random steal if all these checks fail. In addition, at the start of the computation, NABBITC forces a colored steal and each processor may make C checks to find a node of its own color where C may not be a constant.

Lemma 5.3.2. *The total number of colored steals performed by* NABBITC *is* $O(W + S + PC)$ *where* S *is the number of random steal attempts,* W *is the number of steps the processors spend working on computation nodes, and* C *is the number of checks each processor performs at the beginning of the computation. Consequently, the total number of colored steals is bounded by* $O(T_1 + PT_\infty + PM \lg d + P \lg(P/\epsilon) + PC)$

Proof. Trivially, the number of checks at the beginning of the computation is PC since each processor performs at most C of them. After this, after a constant number of checks, a processor has either found work (therefore, these checks are bounded by $O(W)$) or the processor performs a random steal (these checks are bounded by $O(S)$). Summing these up gives us the result. The NABBIT analysis proves that the total number of work steps in the computation is at most T_1 and the total number of steal steps is at most $O(PT_\infty + PM \lg d + P \lg(P/\epsilon))$. This gives us the desired bound. □

At any step, a worker is either working, doing a random steal, or doing a colored steal. Therefore, the total number of processor steps is bounded by $O(T_1 + PT_\infty + PM \lg d + P \lg(P/\epsilon) + PC)$ Since there are a total of P workers, we can divide by P to get the desired running time.

5.4 Design

In this section, we describe our extensions to NABBIT to specify colors, propagate this information through the runtime, and extend the scheduler to be take into account task colors. Throughout

```
1  class DynamicNabbitNode:
2    Key key //this node's key
3    List<Key> predecessors //List of predecessors' keys
4    List<DynamicNabbitNode *> successors //List of successor nodes
5    virtual void init() //initialize this code (user−defined)
6    virtual void compute() //compute this task representing this node (user−defined)
7    int color() = color(node.key) //helper function: this node's color

9  int color(Key key) //user−defined function mapping a key to its color
```

Figure 5.2: NABBITC abstract class interface

this section, we will present the relevant NABBIT interface and the NABBITC extensions for color-aware task graph scheduling.

NABBITC interface

Recall that in NABBIT, users model their computation as a task graph, where nodes are tasks to be computed and edges represent data dependencies between computations. Algorithm 5.2 shows the abstract class interface for defining nodes and their data dependencies. All nodes in a task graph dynamically scheduled by NABBIT inherit from DynamicNabbitNode class and implement the member functions shown in Algorithm 5.2. The init() and compute() functions serve to initialize node parameters and perform the computation represented by the node, respectively. Each task (node in the task graph) is associated with a unique key. The user also specifies the list of predecessors, identified by their keys, this node depends on in the predecessors array. In addition to the information on tasks and their dependencies needed by original NABBIT, NABBITC requires the user to define a color() function that returns a node's color. This function definition serves as the mechanism for the user to provide locality information to NABBITC and is the only additional piece of information the user must provide.

We now present extensions to NABBIT and the underlying Cilk Plus runtime that together constitute the NABBITC infrastructure to exploit this color information to optimize locality.

Designing a locality-guided task-graph scheduler

NABBITC attempts to achieve multiple goals during scheduling: (1) improve data locality by executing nodes of the same color as the executing processor; (2) achieve good load-balance for the computation as a whole; and (3) introduce minimal overhead into the original NABBIT scheduling pathway. Extending the NABBIT task-graph scheduling library to make use of user-provided locality information involves altering how Cilk Plus workers find and determine what to work on. We introduce two specific changes to NABBIT in order to implement this change in policy: (1) **color-aware scheduling using morphing continuations** allow workers to reorganize work so that they may preferentially execute nodes that have the same color as theirs and (2) **colored steals** allow Cilk Plus workers to find work of their color from the current set of stealable frames. In order to implement these policies, we must also change the Cilk Plus runtime system. We now describe these changes.

Color-aware execution order using morphing continuations. The primary source of concurrency in NABBIT is the concurrent processing of all predecessors or successors[4] of a given node. As explained in Section 5.2, NABBIT enables this by spawning the execution of all predecessors (or successors) in parallel using a parallel for loop. NABBIT is oblivious to the order in which these nodes are processed. NABBITC, however, extracts colors from this list of nodes in order to preferentially process nodes with colors that would improve locality.

[4]As described in Section 5.2, NABBIT implicitly maintains successors array for each node u and NABBIT may push successor nodes into it when they must wait for u to complete.

The crucial function for this purpose is the function spawn_colors shown in Algorithm 5.3. This function is called on a list of colors colors (and implicitly, a set of nodes which have these colors). At a high-level, when a processor with color c_p is executing this function, it tries to execute the nodes with color c_p by recursively calling spawn_colors on the half of the list that contains c_p. Once it reaches the base case (the set colors contains only one color), it then spawns all the nodes of color c_p using the function spawn_nodes. This function spawn_nodes is essentially a parallel-for loop over the nodes of this color.

The function spawn_colors essentially re-organizes the order in which nodes are spawned so that the nodes of the preferred color c_p are spawned first. Therefore, it implements what we call a *morphing continuation*. The particular strand that is spawned and the continuation of the strand depends on the color of the processor which is doing the spawn. Another important thing to note about this code is that if the preferred color c_p is not present in the list, the function will spawn the nodes in the original ordering of the list — therefore, a worker does not stall even if it can not find the work of its color.

The function spawn_colors is called in three places in the NABBITC library. Algorithm 5.4 shows the actions to initialize and execute a node. init_and_compute() acquires the colors of the current node's predecessors and invokes spawn_colors() if there exist more than one. Similarly, when spawning the list of successors, compute_and_notify() collects the set of colors for the list of successors and invokes spawn_colors() if there are more than one. Finally, spawn_colors is a recursive function which is also called by itself.

This morphing continuation design allows us to use the same mechanism in two scenarios. First, when a processor spawns the predecessors (or successors) if the node it is currently working on, it uses spawn_nodes to preferentially execute the predecessor(s) (or successor(s)) of its own color. Second, and the more subtle point, is as follows. Note that in Cilk Plus, when a thief worker steals

a task after the spawn of a function[5], it executes the function's continuation. Since spawn_colors is recursive, when a worker steals a continuation, the first statement is executes is spawn_colors. Therefore, the thief also preferentially executes the nodes of *its* color using the same mechanism.

Colored Steals: When a worker has no assigned work, either because it has run out of local work or is at the start of execution, we want that worker to acquire work of its preferred color if possible. In order to do so, we change the stealing policy of Cilk Plus to allow *colored steal* where a worker checks a deque and only steals the work (continuation) at the top of the deque if that continuation contains some node of this worker's preferred color. We will describe the implementation below — we first describe our policy details about when we do colored steals vs. random steals.

One of the goals of NABBITC is to strike a reasonable balance between locality — workers preferentially execute work of their color — and load balance — workers are not idle for too long. In order to do so, we make two changes to the standard Cilk Plus policy of random steals. First, when a worker p with color c_p runs out of work, it does a constant number of colored steal attempts before attempting a random steal. That is, it randomly picks a victim worker q and checks if the frame on the top of q's deque has any tasks of color c_p — if so, it steals this frame making this a successful colored steal. If not, it tries again. If it fails on a constant number of colored steal attempt, it makes a random attempt where it steals whatever is on the top of the victim worker's deque regardless of whether it has a task of color c_p or not. This policy makes sure that p tries to find work of its own color, but then also maintains provable load balance guarantees (as shown in Section 5.3) by greedily doing any work available if it can not easily find work of its color.

There is an exception to this policy, however, at the beginning of the computation. At the beginning of the computation, one worker starts out with executing the root node and all other workers are stealing. At this time, if a worker begins execution in a region of a task graph with no tasks of

[5] A task is represented at runtime by the task's stack or activation frame

its preferred color, it will continue executing the available non-preferred tasks until all work is exhausted (as explained in the morphing continuations section). In addition, often, the first steal represents a significant amount of work (conceptually corresponding to nodes higher up in the task graph or computation tree) and a random first steal can potential to lead poor locality. Therefore, we enforce that the first steal a worker performs is a successful colored steal. After the first steal, the worker follows the policy explained above. This enforcement does affect Cilk's time bound, which we explore in Section 5.3. In our experiments, we found that if all colors are available at the root of the task graph, this time to first work (successful steal) is agnostic to the application, is strictly determined by the number of processors, and, in general, has a small impact on the overall execution time.

We now describe the changes made to both NABBIT and Cilk Plus in order to implement the colored steal policy.

Color-aware GCC Cilk Plus runtime

We make the GCC Cilk Plus runtime color aware by making the following changes. First, we add two additional functions to the Cilk Plus API, shown in Algorithm 5.5, that allows NABBITC to provide color information to the runtime system. The first function is straight forward and is simply used by each worker to set the color of this worker. We pin worker threads and assign them a unique color based on their thread id. The second one is used to implement colored steals and requires more explanation. Recall that in order to do colored steals, a thief worker must be able to tell which color nodes are available in the frame that is on the top of victim worker's deque. This API allows NABBITC to pass this information to Cilk Plus runtime. In particular, before every cilk_spawn, NABBITC calls cilkrts_set_next_colors() to inform the Cilk Plus runtime about which colors are available in the continuation.

The Cilk Plus runtime is also changed with respect to what it does on spawns. At each spawn, the vanilla Cilk Plus pushes the frame of the currently executing function into a worker's deque — allowing some other worker to steal the continuation of the spawn. To enable colored steals, we maintain a color deque alongside the work deque to hold the colors available in each continuation. When NABBITC calls cilkrts_set_next_colors with a set of colors before the spawn statement, this set of colors is pushed at the bottom of the color deque — therefore, each continuation on the work deque has a corresponding set of colors on the color deque.

Now it is easy to see how one can implement colored steals. When a worker p wants to do a colored steal, it simply checks to see of color c_p (p's preferred color) is in the set of colors on the top of victim's color deque. If so, it pops the top of both the color deque and the work deque and puts them on the top of its corresponding deques making it a successful colored steal. Since the number of colors is determined by the number of workers, we make each entry in the colored deque a fixed length array of boolean flags indicating colors contained in the corresponding continuation. This makes the thief's check a constant time operation.

Setting continuation colors in NABBITC As mentioned above, NABBITC must set colors of continuation at each spawn using cilkrts_set_next_colors function. This is done on Lines 12, 29, etc within the code in Figure 5.3. Note that this fits in seamlessly with the design of morphing continuations. At each spawn, we know exactly which colors are available within the spawn and which are available within the continuation. Therefore, NABBITC can easily notify Cilk Plus of the colors available in the continuation by simply telling it which colors are available in the second call to spawn_colors.

Optimizing locality through coloring

NABBITC requires that the user intentionally distribute data across their system and provides a coloring that captures computation locality. We rely on the user knowing how best to distribute data (but not partitioning work among threads), although in many cases an even distribution is sufficient. The coloring the user provides to NABBITC is intended to capture the locality of work performed, based on their data initialization. For this we make two assumptions about color: (1) that data initialized by each individual worker thread is given a unique color and (2) that each node of the computation task-graph is assigned a single color. Requiring the user to describe each node with a single color can lead to some information loss about a node's locality. For example, a node (corresponding to a task) can require data from multiple colored regions and a single color cannot comprehensively describe the node's locality. In these scenarios, the user specifies the node's color to be the one that maximizes locality for that node.

5.5 Experiments

In this section, we evaluate NABBITC by comparing it's performance against original NABBIT and OPENMP. In particular, we try to answer the following questions:

- How well does NABBITC address locality deficiencies in NABBIT? We answer this question using benchmarks in which locality-optimized and load balanced schedules can be created using static scheduling of OPENMP and find that NABBITC provides much better performance than NABBIT and performance comparable to OPENMP.
- How well can NABBITC improve data locality while preserving the dynamic load balancing benefits from NABBIT? We answer this question using the PageRank benchmark, which

cannot be easily statically scheduled, using different data sets. In this case, NABBITC really shines and performs better than both OPENMP and NABBIT.

- To what extent does NABBITC improve data locality? We find that NABBIT has significantly fewer remote accesses compared to NABBIT.
- Does the use of colored steals increase the overall cost to find work as compared to random stealing? We find that while the cost of enforcing the first colored steal is significant, NABBITC makes up for this overhead by having fewer steal attempts later.
- What is the impact of the choice of colors by the user? We consider the behavior of NABBITC using two particularly bad color choices and compare its behavior with NABBIT.

In general, NABBITC shines on benchmarks with irregular memory access patterns, remains competitive with OPENMP when memory accesses are more regular, and almost always outperforms original NABBIT. We observe that our modifications to NABBIT and Cilk Plus introduce minimal overheads, affording performance gains due to a reduction in remote memory accesses when a good coloring is provided.

Experimental Setup All our experiments were performed on an 80-core NUMA machine with 8 Intel Xeon E7-8860 2.27GHz 10-core processors and 1TB of collective DRAM. The machine uses Red Hat Linux 4.4.7-9 configured with 4KB pages. We use a stable GCC 4.9.0 build from the gcc-cilkplus branch for compiling our OPENMP and NABBIT benchmarks and extend this build for NABBITC.

Benchmarks and Baselines We will compare NABBITC performance to NABBIT and OPENMP. OPENMP offers multiple scheduling strategies for parallel for loops. The OPENMPSTATIC policy

simply divides up the iteration space evenly among workers while OPENMPGUIDED dynamically load balances using adaptive block sizes.

Table 5.1 details the benchmarks and input configurations used. We selected various memory-bound applications to demonstrate the importance of achieving good locality when scheduling task-graph computations. The first five benchmarks exhibit regular memory access patterns. We consider these benchmarks to demonstrate the limitations of a dynamic task graph scheduler such as NABBIT that does not account for locality, and evaluate the potential for NABBITC to address these limitations. For these benchmarks, OPENMPSTATIC performs very well if we match the initialization and computation loops; as explained later, this strategy provides optimal locality to regular applications even without locality hints. Therefore, we only compare against this OPENMP strategy since it always performs better than OPENMPGUIDED.

PageRank iteratively computes the PageRank using the power method [60]. This benchmark exhibits access patterns dependent on the graph structure, with varying amounts of work per vertex. We consider three data sets from web crawls [46] that vary in size and graph structure. Specifically, twitter-2010 shows wider variation in its connectivity (e.g., much larger maximum out-degree) than the other data sets considered. On this benchmark, we compare against both OPENMPSTATIC and OPENMPGUIDED strategies for this benchmark.

The Smith-Waterman dynamic program [67] benchmarks exhibit highly regular memory access patterns. We have implemented the wavefront computation in OPENMP, which must synchronize at each diagonal step. In NABBIT and NABBITC, we model the entire computation as a task-graph, exposing more parallelism.

Coloring strategy In all benchmarks, we used OPENMP to distribute data evenly across the machine, with each processor core initializing a unique region of the data. Each thread is pinned to

a processor core and given a unique color. During initialization, each data region is colored based on the color of the thread that initializes it. For regular benchmarks, we group the data accessed by each node based on their color, and pick the color corresponding to the largest fraction of data as the node's color. This color function, provided by the user, can be implemented efficiently for regular benchmarks. Computing the largest color is expensive for irregular benchmarks such as PageRank, where the accesses are data-dependent and involve a large number of irregular accesses. In PageRank, each task takes a block of pages as input, which are accessed regularly, and updates the ranks of pages linked to them, which are accessed in an irregular fashion. The irregular accesses while traversing the links are not avoidable. Therefore, we color each task based on the block of pages it takes as input.

5.5.1 Overall performance

We now demonstrate the effect of locality-guided scheduling on the overall performance. In Figure 5.6, we present the speedup achieved by OPENMP, NABBIT, and NABBITC over serial execution. Error bars show standard deviation across five runs. In general, NABBITC outperforms NABBIT when the problem is sufficiently large. NABBITC shines best with larger irregular PageRank benchmarks, where the impact of locality is more prominent, while remaining competitive with OPENMP on the stencils and NAS benchmarks and outperforming OPENMP for the Smith-Waterman dynamic programs.

We see that in cg, when there are very few nodes in the task graph, NABBITC's benefit over original NABBIT becomes negligible because processor cores have few nodes to work with. With mg, heat, fdtd, and life, when there are many nodes in the task graph, NABBITC is able to continue getting good performance while original NABBIT suffers due to its locality-obliviousness. For these benchmarks, we see that OPENMP consistently performs best. When threads are pinned

and the computation loops are scheduled in the same way as the data initialization loops, OPENMP achieves the maximum locality possible despite not having received any explicit locality hints from the programmer. In addition, it also achieves good load balance, since each iteration does approximately equal amount of work. For these benchmarks, NABBITC's performance approaches that of OPENMP, whereas NABBIT's scalability suffers with increase in core count. For PageRank, OPENMP is not able to maintain its consistency in performance because it is no longer able to achieve locality and load balance simultaneously due to the irregular nature of this application. We see that for larger problems (indicated by the problem size and the larger serial execution time), NABBITC scales better than original NABBIT, OPENMPSTATIC, or OPENMPGUIDED. For Smith-Waterman we see that with the unavoidable remote accesses inherent in the algorithms, NABBITC and NABBIT perform comparably. Both, however, are able to exploit more parallelism than the wavefront OPENMP implementation and edge out ahead.

5.5.2 Locality impact of NABBITC's scheduling strategy

We now look closer at the locality achieved by NABBITC during the execution of these benchmarks. Because counting each memory reference might be expensive[6], we perform this check at the node level in the task graph. This consists of two parts. Note that each of our evaluation system consists of eight NUMA domains, each with 10 cores. First, for each thread, we count the number of nodes it executes that are not the same color as any thread in the same NUMA node. Second, for each thread, we check all predecessors of executed nodes, and count those that are not the same color as any thread in the same NUMA node. Sum of these counts across all threads is reported as the number of remote accesses. For the regular benchmarks, we can compute this as the benchmarks execute without perturbing the execution. For PageRank, this instrumentation can

[6]We were limited by OS version and available hardware counters and were unable to measure remote accesses, stall cycles, etc.

significantly perturb the execution time. Therefore, we track the nodes executed by each thread to record the schedule used in the timing runs. This schedule is replayed to compute the percentage of remote accesses.

Figure 5.7 shows the percentage of accesses that are remote for NABBIT, NABBITC, and OPENMPSTATIC, on 20 or more processor cores (smaller core counts fit in one NUMA domain and do not incur remote accesses). Because NABBIT relies on the random steals in Cilk Plus to disseminate work, the percentage of remote accesses increases with scale, ranging from 45% to 88%, exhibiting a consistent trend across all benchmarks. The introduction of colored steals significantly decreases the percentage of remote accesses. For all benchmarks except twitter-2010 and the Smith-Waterman benchmarks, NABBITC incurs 0% to 9% remote accesses. Importantly, unlike in the case of NABBIT, this percentage does not strictly increase with scale for the regular benchmarks. All strategies incur a high percentage of remote accesses for twitter-2010 and Smith-Waterman.

For regular applications, OPENMPSTATIC incurs almost no remote accesses, as we expect from how the data is initialized. For PageRank, OPENMPSTATIC still has fewer remote accesses than NABBITC; however, as we saw above, it does not have good performance since it is unable to provide good load balance. This result indicates the importance of both locality and load balance—while NABBIT provides great load balance and OPENMPSTATIC provides great locality, NABBITC performs better than both on this irregular benchmark since it simultaneously considers both metrics.

5.5.3 Overheads due to colored steals

The two sources of overhead for NABBITC arise from requiring a constant number of colored steals before performing random steals and forcing the first steal to be a colored steal.

Effect on total steals We now look at the comparison of NABBITC and NABBIT at a more fine-grained level. In Figure 5.8 we see that NABBITC, perhaps counter-intuitively, performs far fewer total successful steals than NABBIT. The introduction of colored steals, and specifically enforcing the first colored steal, helps to significantly reduce the total number steals by ensuring that thieves acquire nodes higher up in the task graph to start with. Due to the depth-first nature of the scheduler, nodes higher up in the task graph have more potential work. Therefore, by ensuring thieves begin with nodes connected to the root, NABBITC is able to effectively increase the amount of work each worker begins with, reducing the total number of steals required.

Overhead due to enforcing first colored steal To calculate the overhead of ensuring that the first steal is a colored steal, Fig. 5.9 shows the average amount of time processor cores spent waiting to acquire work for the heat benchmark. We observed that the times were very similar for all other benchmarks and do not present them here due to space limitations. While this overhead can be substantial, it is agnostic to the application, provided there is at last one node from each color connected to the root. This startup cost can be amortized out with larger, longer running benchmarks. Additionally, recall that we observed that, in practice, enforcing the first colored steal results in far fewer total number of steals which makes up for this overhead.

5.5.4 Importance of good coloring

Overheads with invalid coloring To evaluate this worst case overhead from attempted colored steals, we assigned all nodes an invalid color (no worker has this color), ensuring that all colored steals fail. Therefore, this version of NABBITC behaves like original NABBIT apart from incurring the overheads of colored steals. In Table. 5.3, we see that NABBITC with this alternative coloring performs comparable to original NABBIT indicating that the additional work performed by colored

steals introduces minimal overhead. Specifically, we observe that the mean speedups are within one or two standard deviations, indicating that, for the benchmarks considered, colored steals incur no statistically significant overhead[7].

Behavior under bad coloring The performance of NABBITC is directly tied to the coloring provided by the user. NABBITC assumes the user has constructed a "good" coloring and makes decisions based on this assumption. In the event that the user has provided a "bad" coloring, NABBITC can perform as badly, or worse, than original NABBIT. To test this, we create a coloring where all nodes are given valid incorrect colors. Therefore, in this implementation, all workers will preferentially do non-local work. In Table. 5.2, we see that NABBITC with a bad coloring loses all the performance benefits achieved due to coloring and performs similar to NABBIT. Interestingly, we observe that the mean speedups are within two standard deviations, indicating that NABBIT's locality behavior under random stealing is statistically no better than that of NABBITC under an intentionally bad coloring.

5.6 Related Work

Static task graph schedulers [45, 52, 75] minimize completion time while maximizing locality [72] by completing expanding and analyzing a task graph, together with accurate information on computation and communication costs associated with each task. We consider task graphs that are dynamically explored and do not require prior knowledge of task and communication times.

Cilk's random work stealing is agnostic of locality considerations [33]. Several efforts have incorporated locality considerations by altering the work stealing strategy [37, 51, 55, 63]. These

[7] We are working to understand the characteristics of the uk-2007-05 dataset that make it particularly vulnerable to invalid or bad coloring.

approaches do not naturally extend to scheduling data-flow graphs while preserving provably efficiency in terms of scheduling overheads and effectiveness of load balancing.

Event-driven scheduling strategies map tasks to locality domains together with efficient identification and tracking of ready tasks that can be scheduled [20, 26]. In these systems, data distribution implies a computation partitioning with no further migration of tasks to tackle load imbalance.

SuperMatrix [25], a runtime scheduling system for algorithms operating on blocks as observed in linear algebra programs, mimics a superscalar microarchitecture's scheduling strategy in software. StarPU [9] is a task-graph scheduler for heterogeneous multi-core systems. Neither approach accounts for data locality. Dague [21], a distributed DAG engine, improves locality by working on the local queue when possible. XKaapi [35] is a work-stealing-based scheduler for task graphs that pushes tasks to processors that have better locality for those tasks. It does not preserve the critical path length or provide provable parallel efficiency.

SMPSs [61] schedules dependent tasks together to improve locality. Legion [10] exploits user-specified locality information and coherence properties to perform locality-aware scheduling using a software out-of-order processor. CnC [23] allows the specification of task graphs that are scheduling using a variety of strategies. Legion and CnC also allow user-specification to control task mapping and scheduling (using mappers in Legion and tuners in CnC). Olivier et al. developed various strategies to schedule OpenMP tasks including hierarchical scheduling, and work stealing by one thread on behalf of others in the same chip [58]. None of these schedulers in these systems attempt to preserve optimality guarantees. However, the scheduling strategy developed in this paper can be used to develop provably efficient and locality-aware scheduling algorithms for these task-graph frameworks.

Bugnion et al. [24] developed compiler-directed page coloring techniques to minimize conflict misses. Chilimbi and Shaham [28] identified *hot* data streams and then colocated them to improve

spatial locality. Chen at al. studied scheduling threads for constructive cache sharing [27]. Various approaches have studied the partitioning of shared caches among threads (e.g., [57, 64, 70]). These approaches cannot be applied to optimize NUMA locality considered in this paper.

5.7 Conclusions and Future Work

In this chapter we presented NABBITC, a flexible and easy-to-use task graph library that allows the user to provide locality hints via the use of coloring and provides good load balance via dynamic scheduling. NABBITC is geared towards scheduling on NUMA hardware, where remote accesses may be considerably more expensive than local accesses, but one must strike a balance between locality and load balance to get good performance. Experimental results indicate that this approach is promising, especially for memory intensive irregular applications running on NUMA machines, where static scheduling can compromise load balancing and locality-unaware dynamic scheduling has too many remote accesses. While NABBITC uses Cilk Plus as the underlying language and runtime, we believe this approach can be implemented on other systems such as Intel's Threading Building Blocks.

There are a number of potential directions for future work. Although coloring functions are not often difficult to produce, user can still provide bad, or invalid, colorings which prevent NABBITC from getting any benefit over NABBIT. Providing the user containers which could automate data distribution would allow for the discovering colors automatically, reducing the user's programming effort and ensuring a good coloring is always found.

Allowing for more complex colorings, like hierarchical colors, would allow NABBITC to model the system's memory hierarchy more accurately. Processors could be assigned unique colors to model their cache while those grouped on the same NUMA domain would share a parent color.

Remote memory accesses can also have differing access times depending on the machine's memory network which could be captured with further depth within a coloring hierarchy.

```
1  //Recursively spawn colors using morphing continuations
2  void spawn_colors(colors):
3    if len(colors)==1:
4       spawn_nodes(colors[0])
5    else
6      c_p = /*this worker's color*/
7      /*split available colors into two halves*/
8      first_half = colors[0:len(colors)/2]
9      second_half = colors[len(colors)/2:]
10     if c_p in second_half.keys():
11       swap(first_half, second_half)
12     cilkrts_set_next_colors(second_half.keys())
13     cilk_spawn spawn_colors(first_half)
14     spawn_colors(second_half)
15     cilk_sync

17 //Recursively spawn nodes of the same color
18 void spawn_nodes(nodes):
19   if len(nodes)==1:
20     if nodes[0] is a successor:
21       if nodes[0] is ready:
22         nodes[0].compute_and_notify()
23     else: /*predecessor key*/
24       try_init_compute(this,nodes[0])
25   else:
26     color = nodes[0].color() /*all nodes have same color*/
27     first_half = nodes[0:len(nodes)/2]
28     second_half = nodes[len(nodes)/2:len(nodes)]
29     cilkrts_set_next_colors(color)
30     cilk_spawn spawn_nodes(first_half)
31     spawn_nodes(second_half)
32     cilk_sync
```

Figure 5.3: Pseudo-code for color-aware spawning of a set of nodes in NABBITC using morphing continuations. We use a hybrid C++/Python syntax to enhance readability.

```
1  /*Helper functions to obtain colors*/
2  int color(DynamicNabbitNode node):
3    return node.color()
4  int color(DynamicNabbitNode *node):
5   return node−>color()

7  /*Gather list of spawns based on their color.
8    T = Key (for predecessor list) or
9    T = DynamicNabbitNode (for successor list)*/
10 auto gather_colors(T nodes):
11   //group nodes based on their colors
12   Map<int,List<T>> colors
13   for n in nodes:
14     colors[color(n)].add(n)
15   return colors

17 /*Initialize this (already created) node and compute*/
18 void init_node_and_compute():
19   this.init()
20   colors = gather_colors(this.predecessors)
21   spawn_colors(colors)
22   if all this.predecessors have been computed:
23     this.compute_and_notify()

25 /*Try to initialize node's predecessor with key pkey */
26 void try_init_compute(node, pkey):
27   //atomically attempt to create a predecessor with key pkey
28   pred = /*reference to node for key pkey*/
29   if /*creation succeeded*/:
30     pred.init_node_and_compute()
31   else: /*already created by this or some other thread*/
32     atomic pred.successors.add(node) //enqueue

34 /*compute a node and notify its successors*/
35 void compute_and_notify():
36   this.compute()
37   while /*there are new successors in this.successors*/:
38     colors = gather_colors(this.successors)
39     spawn_colors(colors)
```

Figure 5.4: Key routines to spawn predecessors and successors in NABBITC.

```
1  void cilkrts_set_worker_color(int color)
2  void cilkrts_set_next_colors(List<int> colors)
```

Figure 5.5: Extensions to the Cilk Plus RTS API to inform the runtime of the worker's preferred color and the colors available in a continuation.

Benchmark	Description	Problem size	Iterations	Task graph nodes	Serial time (seconds)
cg	NAS conjugate gradient	$NA = 900000$, $NNZ = 26$	1	300	309
mg	NAS multigrid	$n\{x, y, z\} = 2048$, $LM = 11$	1	16384	690
heat	Heat diffusion stencil	$n = 16384$, $m = 655360$	5	102400	377
fdtd	Finite difference time domain	$n = 16384$, $m = 655360$	5	102400	970
life	Conway's game of life	$n = 16384$, $m = 655360$	5	102400	275
page-uk-2002	PageRank uk-2002 dataset	$nv = 18$M, $ne = 298$M	10	1800	198
page-twitter-2010	PageRank twitter-2010 dataset	$nv = 41$M, $ne = 1468$M	10	4100	1025
page-uk-2007-05	PageRank uk-2007-05 dataset	$nv = 105$M, $ne = 3738$M	10	10500	900
sw	Smith-Waterman (n^3)	$\{n, m\} = 5120$, $B = 32x32$	1	25600	450
swn2	Smith-Waterman (n^2)	$\{n, m\} = 131072$, $B = 1024x1024$	1	16384	179

Table 5.1: Benchmark configurations and serial OPENMPSTATIC execution time. The PageRank benchmarks use the same code with three different web crawl datasets [18, 19, 46].

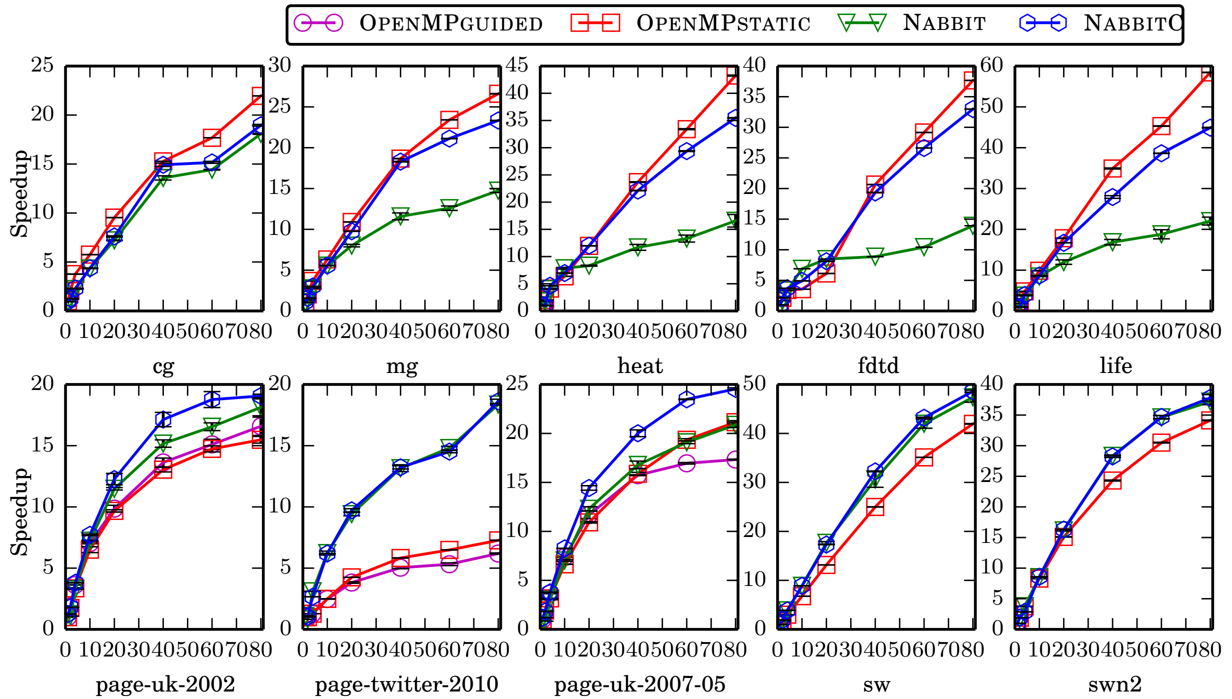


Figure 5.6: Speedup for all benchmarks. x-axis: number of threads (processor cores); y-axis: speedup over serial OpenMPStatic.

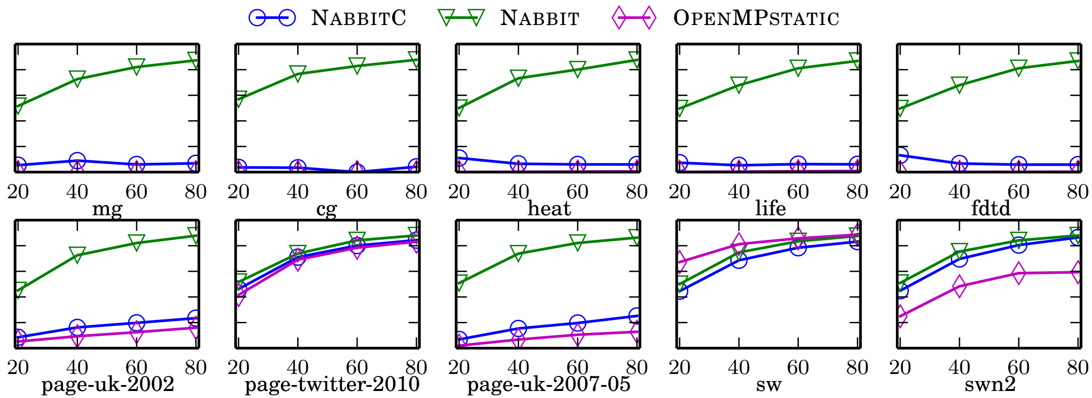


Figure 5.7: Percentage of accesses that are to data in remote NUMA domains. We show percentages for 20–80 cores (1–10 cores fit in one NUMA domain and do not incur remote accesses). x-axis: core count; y-axis: Percentage of accesses that are remote.

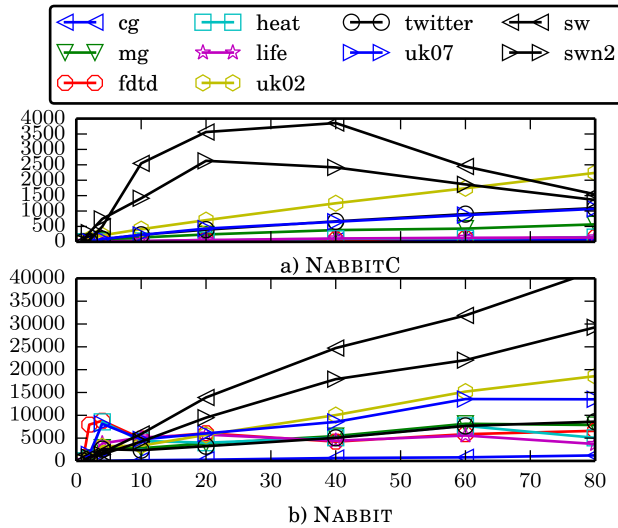


Figure 5.8: Average number of successful steals for (a) NabbitC and (b) Nabbit. x-axis: number of cores; y-axis: Average number of successful steals

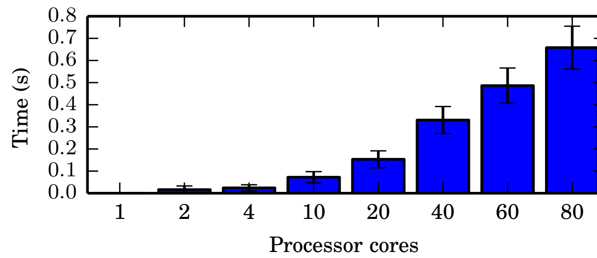


Figure 5.9: Average idle time per processor core (across all processor cores and runs) due to forcing the first colored steal for the heat benchmark. Error bars show standard deviation across five runs among all processor cores. We observed this time was the same for all benchmarks.

P	cg	mg	heat	fdtd	life	uk-02	twitter	uk-07	sw	swn2
20	0.96	0.76	0.66	1.08	0.65	0.75	0.83	0.76	0.76	0.66
40	0.97	0.88	0.78	1.10	0.77	0.83	0.55	0.94	0.88	0.78
60	0.96	0.97	0.95	1.08	0.94	0.89	0.49	1.02	0.97	0.95
80	1.02	0.96	0.93	0.94	0.97	0.91	0.41	1.02	0.96	0.93
	0.98	0.89	0.83	1.05	0.83	0.85	0.57	0.93	0.89	0.83
	0.02	0.08	0.12	0.06	0.13	0.06	0.16	0.11	0.08	0.12

Table 5.2: Speedup of NabbitC over Nabbit when all tasks are assigned bad colors resulting in preferential execution of non-local tasks. S.D. denotes standard deviation.

P	cg	mg	heat	fdtd	life	uk02	twitter	uk07	sw	swn2
20	1.03	0.99	0.94	1.06	0.93	1.03	1.12	1.12	0.99	0.94
40	1.02	0.99	0.99	1.04	0.92	0.97	1.09	1.10	0.99	0.99
60	0.99	0.98	0.94	1.03	0.92	0.98	1.01	1.08	0.98	0.94
80	1.06	0.97	0.88	0.91	0.98	0.94	1.07	1.07	0.97	0.88
	1.03	0.99	0.94	1.01	0.94	0.98	1.07	1.09	0.99	0.94
	0.02	0.01	0.04	0.06	0.03	0.03	0.04	0.02	0.01	0.04

Table 5.3: Speedup of NABBITC over NABBIT when all tasks are assigned invalid colors resulting in failure of all colored steal attempts. S.D. denotes standard deviation.

Chapter 6

Locality-Friendly Parallel For Loops

6.1 Introduction

PARALLEL_FOR loops are a simple mechanism for programmers to understand: rather than execute sequentially, loop iterations can instead be executed in parallel. Programming languages that offer a PARALLEL_FOR loop typically make use of the same syntax as a regular loop, making it easy for programmers to parallelize their applications with minimal changes to their code.

There exist many applications where the core of the work done is within the iterations of a for loop all of which can often benefit directly from PARALLEL_FOR loops, provided there does not exist any data dependencies between iterations.

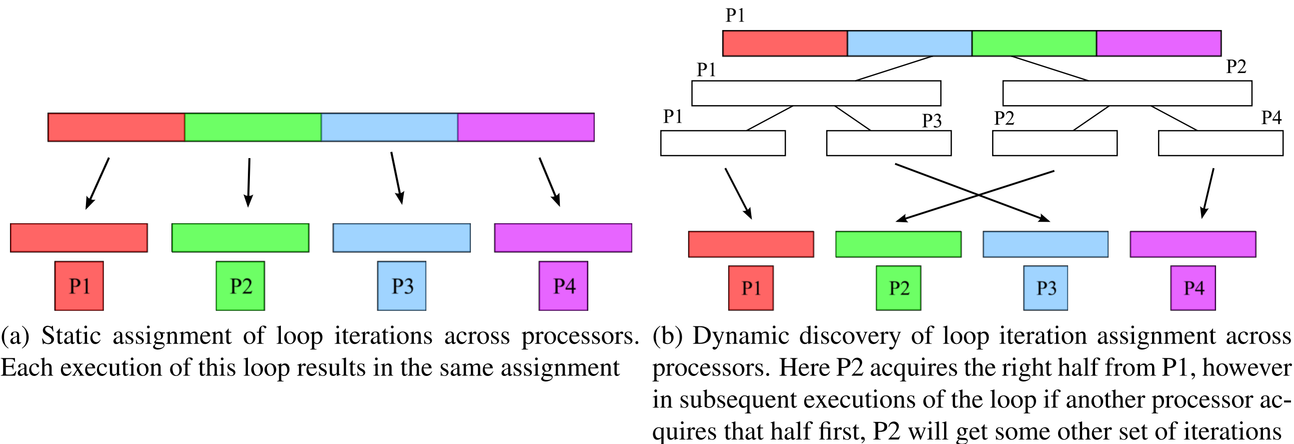
In many cases where an application utilizes a series of PARALLEL_FOR loops, such as performing the same computation over many time-steps, if the same cores execute the same iterations each invocation of the loop, they end up reducing cache misses as well as remote memory accesses when executing on a machine with non-uniform memory accesses. Figure 6.1a demonstrates how data might get used during a statically scheduled PARALLEL_FOR loop. By assigning iterations

directly to cores, the data needed by an individual core will be the same each time the loop is executed.

Although CILKPLUS, a popular parallel programming language, offers PARALLEL_FOR loops, they are executed dynamically, with no guarantee of which core executes which iteration, preventing them from being able to take advantage of these potential locality benefits. In CILKPLUS, PARALLEL_FOR loops are syntactic sugar for recursively split loop iterations that are scheduled dynamically at runtime. Figure 6.1b demonstrates one iteration of a CILKPLUS PARALLEL_FOR. The iterations acquired by P2 depend on it acquiring half of the total iterations from P1 in the beginning. If another core was able to acquire this instead in in a successive execution, P2 would eventually find a different set of iterations. Although this PARALLEL_FOR implementation does not readily benefit from locality it ensures load balance.

In this chapter we present locality-friendly PARALLEL_FOR loops for the CILKPLUS runtime. Our goals are to:

1. Elevate PARALLEL_FOR loops beyond syntactic sugar into first class citizens within the CILKPLUS runtime.
2. Minimize the changes required of the user to when parallelizing their loops or switching from the current PARALLEL_FOR implementation.
3. Implement statically assigned iterations so that cores may execute the same iterations each time a loop is invoked.
4. Support dynamic load-balancing after initial assignments have been completed.
5. Have as minimal overhead as possible.



(a) Static assignment of loop iterations across processors. Each execution of this loop results in the same assignment

(b) Dynamic discovery of loop iteration assignment across processors. Here P2 acquires the right half from P1, however in subsequent executions of the loop if another processor acquires that half first, P2 will get some other set of iterations

While we find that our design for CILK_PARFOR is indeed able to accomplish most of our goals, it introduces additional overhead which can become problematic for applications with small workloads.

6.2 Related Work

PARALLEL_FOR loops have been implemented in some form or another for many commonly used programming languages. These implementations, do not, however, combine good load-balance while addressing locality.

Two related PARALLEL_FOR scheduling algorithms address locality directly. Markatos and Leblanc [54] proposed the affinity scheduling algorithm which divides the iteration space evenly across cores, has them work on decreasingly smaller subsets of iterations until completed and then allows them to assist the core with the largest remaining set of iterations. Li et al. [49] expand upon AFS by first distributing data across NUMA domains and having workers begin with local work first. They improve temporal locality by considering the data distribution used when ordering iterations execution and reduce the number of synchronization operations by unifying their subtask sizing policy.

While both of these algorithms consider locality, they rely on global synchronization which can introduce significant overhead.

6.3 Background

In this section we provide a brief overview of the CILKPLUS runtime system including the programming interface it provides, the basic data structures used to implement randomized work-stealing and its implementation of PARALLEL_FOR as syntactic sugar.

The CILKPLUS runtime utilizes a randomized work-stealing scheduler to efficiently execute applications in parallel. It houses a number of ***worker*** threads which, in distributed fashion, acquire work from one another to dynamically. Programmers use the cilk_spawn and cilk_sync keywords which allow to mark opportunities for parallelism and enforce synchronization respectively.

At a high-level, a cilk_spawn designates that the spawned routine is *capable* of running in parallel with everything after the cilk_spawn and until the next cilk_sync. It is important to not that a cilk_spawn does not guarantee that the routine will in fact execute in parallel with the following code. Users specify *logical* parallelism while workers determine at runtime what will be executed where.

When a worker spawns a function, it begins executing that function right away and makes the remaining work available for some other worker to steal. This stealable work takes the form of a ***frame*** stored inside the worker's double-ended queue ***deque***. During normal execution, workers push and pop frames from the *bottom* of their deque, in similar fashion to the C call stack. When a worker runs out of work they become a thief and attempt to randomly steal a frame from the *top* of another worker's deque. Frames at the top of a worker's deque are from spawns performed

earlier in the application and have a higher potential for capturing a larger amount of the workload, a feature which becomes important for proving CILK's time bound.

CILKPLUS additionally offers a cilk_for loop which automatically parallelizes the of a PARALLEL_FOR loop. These loops are implemented as syntactic sugar for recursive spawns, making the runtime completely unaware of the cilk_for loops. cilk_for loops are recursively split in half, either to sets of size n/P, where n is the number of iterations and P is the number of workers, or to a user-specified size. At each split, half of the iterations are spawned, allowing a thief to acquire them, while the other is taken by the current active worker.

Due to the fact that this implementation is built with cilk_spawn and cilk_sync, it is able to maintain CILK's optimal time bound. However, in memory bound applications, where locality can play the main role in determining performance, a dynamically scheduled PARALLEL_FOR loop can perform poorly. cilk_for cannot ensure loop iterations are executed by the same worker over multiple invocations, forcing workers to continually require new memory from random locations which can severely impact performance.

6.4 Design

In this section we detail the design of our CILK_PARFOR loops which are no longer just syntactic sugar for a series of spawns but instead first class citizens handled within the CILKPLUS runtime. The main objective of our design is to address the inherent locality issues of successive, dynamically scheduled PARALLEL_FOR loops by introducing static assignments – workers can get better locality by executing the same set of iterations each time a PARALLEL_FOR loop is invoked. Upon completion of their initial assignment, workers are then able to steal iterations from one another sacrificing locality for load-balance.

6.4.1 Assigning Iterations to Workers

Our CILK_PARFOR loops utilize a static assignment policy, directing workers to execute the same subset of iterations each time the CILK_PARFOR is executed. Working on the same subset of iterations for each CILK_PARFOR loop invocation can grant both cache and NUMA locality benefits. Additionally, a naive assignment of iterations can also result in an even load distribution when individual iterations have similar workloads.

To distribute iterations across workers, CILK_PARFOR divides up the iteration space into evenly sized contiguous subsets and passes each to a unique worker. Each worker is given an equal number of iterations and no iteration is given to more than one worker. In the event that there are more workers than iterations, CILK_PARFOR will assign a single iteration to as many workers as there are iterations.

With such an assignment policy, there are a number of ways CILK_PARFOR can get locality benefits. Consider the case where the data required by the subset of iterations assigned to a particular worker is able to fit in that worker's private cache. If the CILK_PARFOR loop is executed again, the worker will be assigned the same iterations as before and therefore the data require will already be in cache. Dynamic scheduling policies can not make the guarantee that workers execute the same iterations during each CILK_PARFOR and as a result, incur far more cache misses.

On NUMA machines, static assignment policies can be particularly good at reducing the number of expensive remote memory accesses. To achieve this, the data used by a particular worker must live in that worker's NUMA domain. User's can infer the iteration assignments that our policy will find and distribute the necessary data accordingly. Assuming first-touch this can also be done with the use of a starting initialization CILK_PARFOR loop at the beginning of the application.

6.4.2 Elevating CILK_PARFOR beyond syntactic sugar

Vanilla CILKPLUS executes cilk_for loops by recursively splitting and spawning iterations, making it simply syntactic sugar for a number of spawns. To elevate CILK_PARFOR loops into first class citizens, we introduce the notion of CILK_PARFOR frames into the runtime system, add a new meta data structure into the runtime to hold the loop information and update the scheduling routines to handle CILK_PARFOR loops accordingly.

We've introduced two special frame types for CILK_PARFOR loops, PARALLEL_FOR parent frames and iteration frames. The PARALLEL_FOR parent frame is marked upon creation of the CILK_-PARFOR loop and will collect child iteration frames as workers arrive and execute their assigned iterations. Iteration frames are not instantiated during the creation of the CILK_PARFOR loop but instead created when a worker begins executing their assignment. Workers discover the CILK_-PARFOR loop by attempting to steal from a victim who's working on the loop. Once discovered, rather than taking work off of the victim's deque, the thief creates a new iteration frame and begins executing their assignment.

The CILK_PARFOR meta data structure holds all of the information pertinent to the CILK_PARFOR loop, including the iteration assignment for each worker as well as a pointer to the loop's body. The iterations within a worker's assignment are the remaining iterations that the worker will execute. Upon creation of the CILK_PARFOR loop, this will be the worker's initial static assignment and as the worker completes iterations, it updates it's current assignment to reflect it's progression.

6.4.3 Stealing iterations towards load balance

Workers may not always complete their iteration assignments simultaneously, either due to imbalance in the workloads of individual iterations or differences in when they start executing their

```
1        CILK_PARFOR_BEGIN(it,start,end,incr) {
2            ...
3        } CILK_PARFOR_END(it,start,end,incr);
```

Figure 6.1: CILK_PARFOR user interface

iterations. To address this problem, we include support to allow workers to steal iterations from one another. Stealing iterations from a victim requires updating their current assignment. Recall that a worker's assignment is it's remaining iterations to execute. As a worker progresses through its assignment, it takes iterations from the "front" while a thief will steal iterations from the "back". To prevent races on iterations we ensure only a single thief can attempt to steal by requiring it to hold the victim's steal lock. The victim is not required to take this lock in order to take iterations, instead we utilize the Dekker protocol [**?**] which ensures that the victim and the single thief do not take the same iterations.

6.5 Implementation

In this section we describe the implementation details of our CILK_PARFOR loops. Our aim is to adopt the classic PARALLEL_FOR programming interface and include runtime support within CILKPLUS for executing PARALLEL_FOR loops natively. Our implementation works alongside the normal CILKPLUS scheduling body, relying on an augmented steal policy to both find assigned work and continue work-stealing when all assigned iterations are completed.

6.5.1 Programming Interface

CILK_PARFOR uses a familiar programming interface, requiring from the user an iterator, an initial assignment, a final assignment and an increment to iterate with. We demonstrate this interface in

Figure 6.1. The programmer utilizes a pair of macros wrapped around their iteration code to specify their CILK_PARFOR. In this signature, it is a variable name which may or may not be instantiated already, while start,end and incr are integers[8] for the initial assignment, the final assignment and the increment value .

Although this interface does not directly mimic the traditional PARALLEL_FOR programming interface, it makes use of the same input parameters. We utilize macro definitions over a novel keyword for ease of implementation as doing so only requires changes to the library and not the compiler itself.

6.5.2 CILK_PARFOR loop execution

CILK_PARFOR loops are executed in three stages: 1) creation, 2) iteration execution and eventually 3) destruction. The user-thread, which we refer to as the **creator**, is responsible for storing the iteration code block, as well as creating and populating a metadata structure to hold assignments and book-keeping utilities. Once finished, the creator begins executing their assigned iterations while other workers begin arrive to do the same. Upon completion of all iterations, control is returned to the creator thread to cleanup and end the CILK_PARFOR.

CILK_PARFOR creation

The creation of a CILK_PARFOR involves two main tasks: 1) storing the user's iteration code block and 2) instantiating a metadata structure to store the CILK_PARFOR parameters and hold relevant book keeping information.

[8]Although we currently only support integers, our design could readily accept arbitrary data types via templating.

```
1  #define CILK_PARFOR_BEGIN(it_n,st,ed,inc) \
2  do { \
3      auto iter_body_spawn_helper = \
4          [&](__cilkrts_parfor_data *__cilk_pfd,__cilkrts_stack_frame *__cilk_psf) { \
5              __cilkrts_worker *w = __cilkrts_get_tls_worker(); \
6              CILK_PARFOR_ITER_PROLOG(__cilk_psf,w) \
7              auto iter_body_func = \
8                  [&](__cilkrts_parfor_data *__cilk_pfd) \
9                  { \
10                     int done = 0; \
11                     __cilkrts_parfor_iter_data __cilk_itd; \
12                     __cilkrts_parfor_cnt_t it_n; \
13                     while(!done) { \
14                         done = !__cilkrts_check_for_iteration(__cilk_pfd,w,&__cilk_itd); \
15                         it_n = __cilk_itd.start * inc; \
16                         while (it_n < __cilk_itd.end*inc) { \
17                             do
18                                 /* Iteration code goes here */
19 #define CILK_PARFOR_END(it_n,st,ed,inc) \
20                             while(0); \
21                             it_n += inc; \
22                         } \
23                     } \
24                     CILK_PARFOR_ITER_EPILOG() \
25                     __cilkrts_end_iteration(w,__cilk_pfd,&__cilk_iter_sf); \
26                 }; \
27              iter_body_func(__cilk_pfd); \
28              }; \
29      __cilk_parfor_execute_loop(st,ed,inc,iter_body_spawn_helper); \
30 } while (0);
```

Figure 6.2: The expansion of the CILK_PARFOR_BEGIN and CILK_PARFOR_END macros

```
1  template <typename IterHelperFunc>
2  CILK_PARFOR_NO_INLINE
3  void __cilk_parfor_execute_loop(__cilkrts_parfor_cnt_t _start,
4                                  __cilkrts_parfor_cnt_t _end,
5                                  __cilkrts_parfor_cnt_t _incr,
6                                  const IterHelperFunc& iter_helper_func)
7  {
8      // We're CALLING P4,
9      __cilkrts_stack_frame __cilk_psf;
10     __cilkrts_parfor_data * pfd;
11     CILK_PARFOR_PROLOG(__cilk_psf);
12     __cilkrts_worker *w = __cilkrts_get_tls_worker();
13     // Get ready to set the jump point
14     CILK_PARFOR_FAKE_SAVE_FP(__cilk_psf);
15     int ret = CILK_SETJMP(__cilk_psf.ctx);
16     if (ret == 0) { // Setup
17         __cilkrts_parfor_params pfp = {_start,_end,_incr};
18         __cilkrts_setup_parfor(&pfd,&(__cilk_psf),&pfp);
19     }
20     __cilkrts_worker *wk = __cilkrts_get_tls_worker();
21     if (pfd->jump_flag[wk->self] <= 1) { // Execute iterations
22         iter_helper_func(pfd,&__cilk_psf);
23         // Worker will never return here
24     }
25     else { // Clean up
26         __cilkrts_destroy_parfor(pfd);
27     }
28     CILK_PARFOR_EPILOG();
29     return;
30 }
```

Figure 6.3: The main CILK_PARFOR execution body

The storing of the iteration code block is handled within the expansion of the provided macros, as shown in Figure 6.2. The user's iteration code is wrapped within a helper lambda function which the creator takes into __cilk_parfor_execute_loop, the main CILK_PARFOR execution body. This helper function will be used by each worker to iterate over their assigned iterations and execute the user's code block with the appropriate iterator setting.

In Figure 6.3 we show __cilk_parfor_execute_loop, the main execution body for CILK_PARFOR. When the creator first executes this function it creates an instance of __cilkrts_parfor_data which

we refer to as the CILK_PARFOR's pfd. The pfd is a shared data structure holding the CILK_-PARFOR parameters, worker's assignments as well as all bookkeeping information. Following this instantiation, the creator calls setjmp to store a jump location which all other workers will utilize to enter the execution body and begin their iterations. A call to longjmp with this jump location results in that worker returning from the setjmp call with a return value of 1 – The creator ,who first calls setjmp, returns with a value of 0. Checking the return value of setjmp allows the creator to complete the CILK_PARFOR setup.

We utilize this jump location for workers to begin executing a set of iterations and for the creator to return and end the CILK_PARFOR. To differentiate between which jump is occurring, we check a jump flag which is always set immediately before exiting the runtime.

Executing iterations

Although the creator is able to begin executing iterations immediately after creating the CILK_-PARFOR loop, there are two tasks which must be completed before any other worker is able to begin their own iterations. Before the first non-creator worker can begin executing iterations, they must first 1) clear out any work from the creator's deque that is above the CILK_PARFOR frame and 2) detach and suspend the CILK_PARFOR frame. Once completed, all workers are able to begin executing their assigned iterations.

The creator's deque must first be cleared before any other worker can start their iterations. When the creator begins executing their assignment, they push a frame associated with the CILK_PARFOR onto the bottom of their deque and proceed to immediately execute their iterations with a new iteration frame. To discover the CILK_PARFOR loop, workers must perform a steal attempt that finds either the CILK_PARFOR frame (in the case that no other have done so yet) or an iteration

frame. Since steal attempts only look at the top of a victim's deque, thieves must first pop all other frames to access the CILK_PARFOR frame.

Once the first worker encounters the CILK_PARFOR frame rather than take the frame for themselves, as is done during a normal steal attempt, they suspend that frame and attach to it a new iteration frame which they create for themselves. Recall that a suspended frame is a frame not currently owned by a worker to which child frames have been attached. A suspended frame is meant to become ready for execution when all of it's children have completed. The parent CILK_PARFOR frame keeps track of the active child iteration frames with it's join counter, a value which will be important for cleanup.

When a worker is ready to begin executing a set of iterations, they set their jump flag within the pfd and make a call to longjmp to return to the location stored by the creator's call to setjmp (6.3, Line 15) and begin executing their assignment. Whenever a worker has jumped to this point with a jump flag of 1, they have an assigned set of iterations ready execute. Each worker's assignment is stored as a set of start and end indices held within the pfd. Upon the creation of the CILK_-PARFOR these indices represent the worker's initial assignment, however, if the worker completes their initial assignment and steals more iterations, the start and end indices will be set to reflect this new assignment – these indices effectively represent the iterations that the worker has left to execute when they next as they arrive and proceed with the CILK_PARFOR execution body. To execute these iterations, the worker calls the helper function holding the user's iteration code.

Lines 13-28 from Figure 6.2 show the loop responsible for executing assigned iterations. While iterations remain, the worker queries the pfd for a new iteration to execute. __cilkrts_check_for_iteration increments the worker's start index by one, [9] indicating that the iteration at that index will now

[9] At the moment we only allow for iterations to be divided into grains size of 1, however, supporting larger grain simply requires changing the amount with which __cilkrts_check_for_iteration increments the start index.

be executed. This continues until there are no more iterations left within the worker's current assignment. To allow workers to steal iterations from each other, __cilkrts_check_for_iteration uses the Dekker protocol, a lock-free resource sharing algorithm, to prevent races on iterations. Thieves are responsible for acquiring a steal lock before attempting to interrupt the victim and acquire iterations through Dekker.

Steal policy

CILK_PARFOR integrates within CILKPLUS's runtime scheduling through an augmented steal policy which allows workers to discover assignments or find iterations to steal. Continuing to utilize steals as the mechanism for acquiring work affords us a natural integration into the scheduling while readily providing support for nested parallelism within iterations. Although such support comes with our implementation we leave the exploration of this feature to future work.

Rather than instrumenting a novel mechanism to pass iterations directly, , workers discover the CILK_PARFOR and the iterations they should execute through random steal attempts. During a normal steal attempt, a worker acquires a frame from the top of the victim's deque and proceeds to execute it. When a thief finds a victim working on CILK_PARFOR iterations, by finding either a CILK_PARFOR frame or an iteration frame, they instead find their own set of iterations to execute. The following priority is observed when a thief searches for iterations upon encountering a CILK_PARFOR during a random steal:

1. The thief's initially assigned iterations
2. An initial assignment which some other worker has yet to start
3. The iterations of a randomly chosen victim

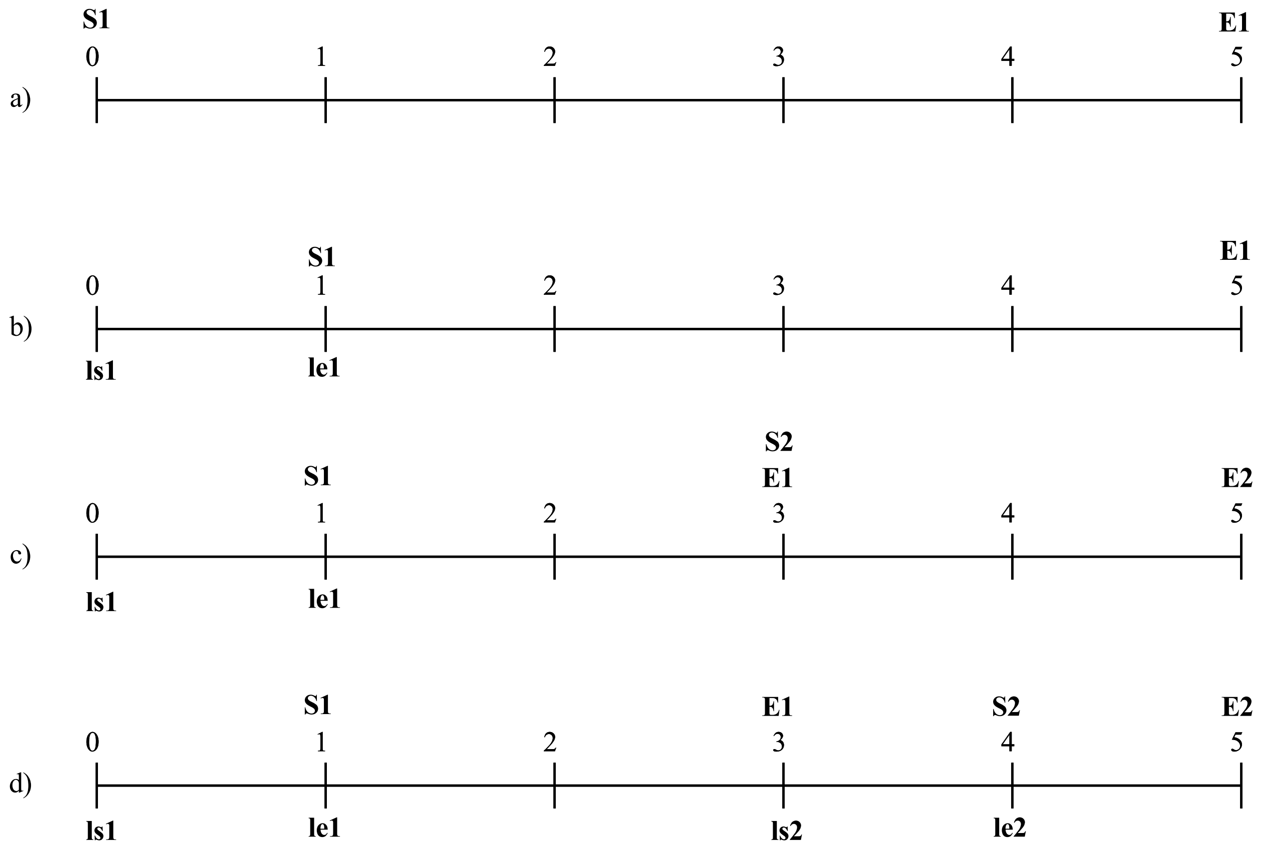


Figure 6.4: Iteration steal example

Prioritizing the initial static assignment ensures the main goal of our implementation, to allow workers to acquire the same iterations over multiple invocations of a CILK_PARFOR. Our secondary goal, to employ dynamic load-balance when initial assignments have finished, is accomplished by allowing workers to steal iterations from one another. When stealing iterations, we prioritize an initial assignment which has yet to start over the iterations of a random victim, ensuring that the theif acquires as many iterations as possible. Allowing workers to steal iterations from themselves sacrifices the benefits of always executing the same iterations to reduce idle time and improve load-balance. In some cases, however, the cost of a worker computing an iteration that was not initially assigned to them could out-weigh the idle time saved from stealing it, we therefore allow users to disable the stealing of iterations to ensure a purely static distribution.

Once a victim has been found, either by discovering they have not begun their initial assignment or via random selection, the thief steals half of the victim's current assignment for themselves. Figure 6.4 demonstrates this.

a) Worker 1 is assigned 5 iterations where $S1$ and $E1$ represent it's start and end indices respectively.

b) Worker 1 takes the gets the first iteration to execute when it calls __cilkrts_check_for_iteration (Figure 6.2: Line 14). $ls1$ and $le1$ are the local start and end values that are used by the grain execution loop Figure 6.2: Lines 16-22.

c) Worker 2 steals iterations from Worker 1. The theif accomplishes this by setting $E2 \leftarrow E1$, $E1, S2 \leftarrow S1 + (\frac{E1-S1}{2})$.

d) Worker 2 begins executing the newly stolen iterations.

Once a worker has updated their starting index, the iterations they have taken are no longer available for stealing – thieves are only capable of stealing from a victim's set of assigned iterations.

Cleaning up

Upon completion of all the CILK_PARFOR iterations, the creator is responsible for destroying the __cilkrts_parfor_data instance and exiting the CILK_PARFOR.

When a worker finishes an assigned set of iterations, they return to the runtime with a call to __cilkrts_end_iteration. There, they unlink their current frame from the parent CILK_PARFOR frame decrementing the parent's join counter. When the join counter is brought to zero there are no more active workers executing iterations, however, it may be the case that some worker has yet

to arrive and begin executing their assigned iterations. To ensure all iterations are completed, a worker decrementing the parent's join counter to zero will also check if there exist any workers who have yet to arrive, and if so, steal iterations from them. If this is not the case, the parent CILK_PARFOR frame is then pushed to the creator, who's jump lag is set so that after returning to the stored jump location, Line 00 in Figure **??**, they finalize the CILK_PARFOR by destroying the __cilkrts_parfor_data and continuing with the remainder of the application.

6.6 Experiments

In this section we present an evaluation of our CILK_PARFOR loops which address key points from our initial goals. Our aim is to show our implementation:

1. Performs no worse than vanilla cilk_for loops on time-bound applications
2. Can improve speedup over cilk_for loops on data-bound applications
3. Introduces minimal overhead

The following experiments will show our CILK_PARFOR loops can achieve goals 1) and 2) but falls short on 3). Our implementation introduce a non-trivial overhead which can result in poor performance in some cases.

6.6.1 Experimental Setup

Our experiments were run on a 32-core machine with four Intel Eight Core Xeon E5-4620 2.2Ghz processors which have 16M shared L3 and 256K private L2 caches. Our experiments were compiled using gcc 5.2.0 using a modified CILKPLUS library for our CILK_PARFOR loops.

We focus on the average overall speedup of the applications over 10 runs and compare the following:

- cilk_for: CILKPLUS' vanilla cilk_for loops
- OPENMPSTATIC: OPENMP's static PARALLEL_FOR loops
- CILK_PARFOR-HYBRID: Our CILK_PARFOR loops with static iteration assignments and iteration stealing enabled
- CILK_PARFOR-STATIC: Our CILK_PARFOR loops with static iteration assignments and iteration stealing disabled

We include OPENMP loops in our evaluation as they provide the type of statically assigned iterations that we want without support for load-balance. To compute speedup, we compare against OPENMP's serial execution time.

6.6.2 Time-Bound Applications

Although time-bound applications are unable to benefit from improvements in locality it is important to ensure our CILK_PARFOR can perform as well as vanilla cilk_for loops. To explore this application type, we implemented the wavefront version of the n^3 Smith-Waterman algorithm. For this experiment we use a problem size of 2048x2048 with blocks of size 16x16.

Figure 6.5 shows speedup over serial OPENMP as we increase the number of cores. All implementations achieve roughly the same speedup over the baseline. In this case, CILKPLUS's dynamically scheduled cilk_for is as good as OPENMP's purely statically scheduled PARALLEL_FOR loops and our CILK_PARFOR loops are no worse than either regardless of whether or not iteration stealing is enabled.

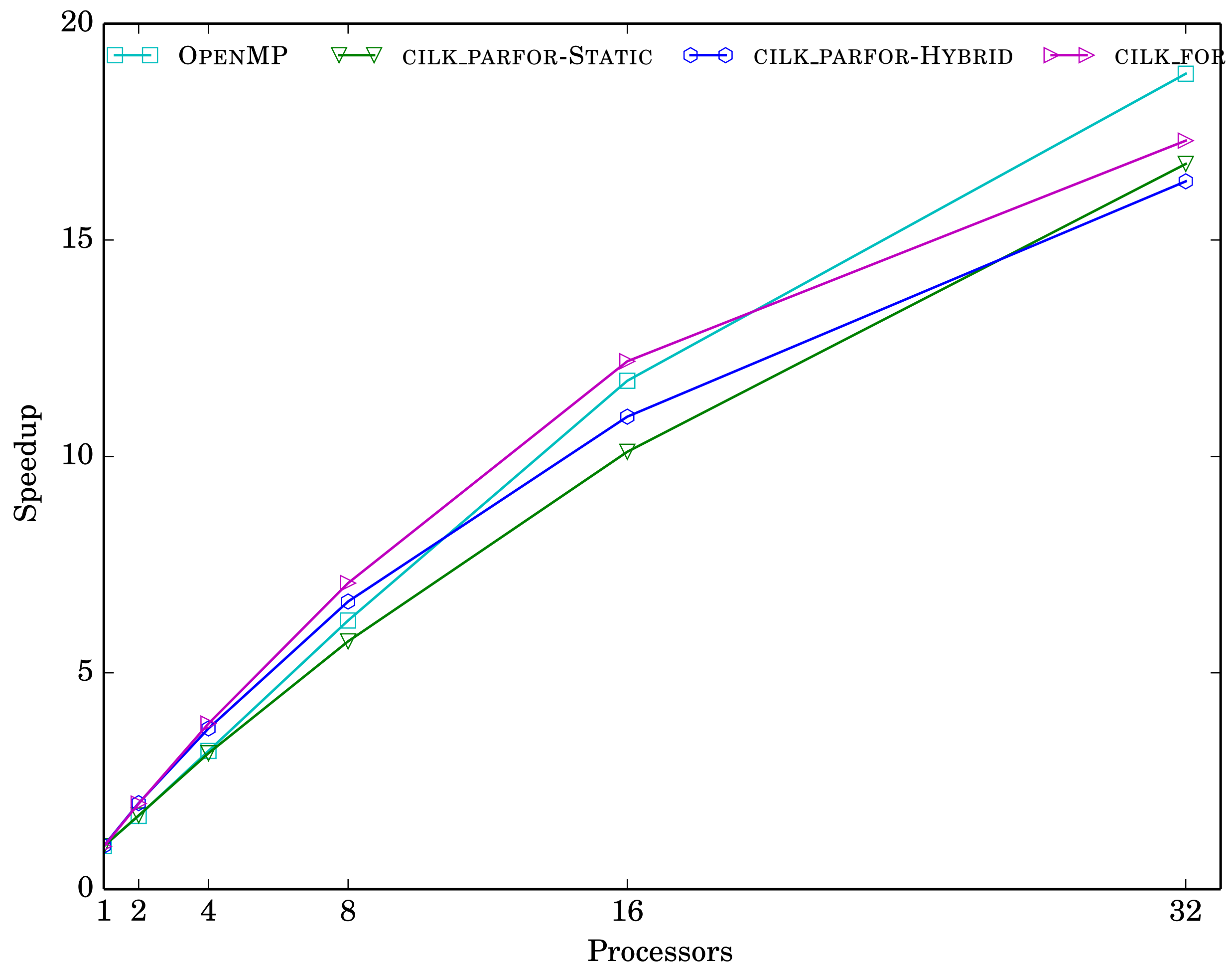


Figure 6.5: Smith-Waterman results

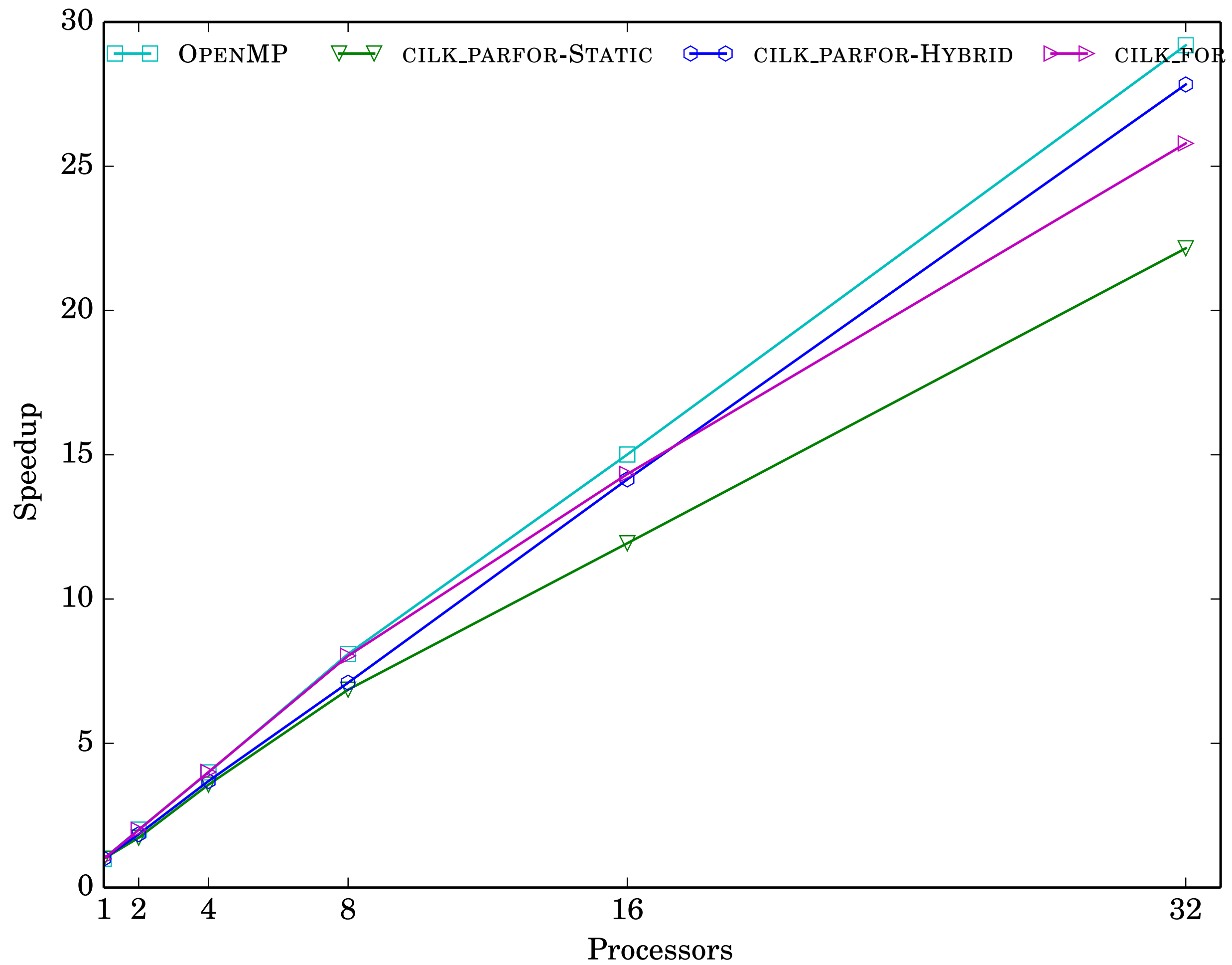


Figure 6.6: Conway's Game of Life results

6.6.3 Data-Bound Applications

Data-bound applications rely more heavily on locality as the majority of their workload comes from performing data accesses. To evaluate our CILK_PARFOR's effect on locality we look a stencil application with very regular memory access patterns. Conway's Game of Life is a cellular automaton which computes whether a cell lives or dies over time depending on it's immediate neighborhood. For this experiment we use a problem size of 16K x 16K and use blocks of size 16x256.

Figure 6.6 shows speedup over serial OPENMP as we increase the number of cores. From these results notice two key observations: 1) OPENMP performs best for this application, 2) CILK_-PARFOR-STATIC performs worse than cilk_for, however enabling iteration stealing in CILK_PARFOR-HYBRID allows our CILK_PARFOR loops to compete with OPENMP.

OPENMP is able to outperform all other implementations on such an application because a strictly static assignment of iterations leads to perfect locality. Furthermore, with the application itself being highly regular, an even distribution of iterations results in an even distribution of work. OPENMP has the added benefit of introducing very minimal overhead, with it's threads being able to begin executing iterations immediately.

Surprisingly CILK_PARFOR-STATIC performs poorly, despite being purely static like OPENMP. The reason for this difference being that CILK_PARFOR-STATIC workers are not able to execute iterations immediately and must first perform random steal attempts in order to discover the CILK_-PARFOR loop. This delay does, however, not explain why CILK_PARFOR-STATIC performs worse than cilk_for. With iteration stealing disabled, workers in CILK_PARFOR-STATIC go idle once they have completed their assignment. When coupled with delayed starts, this results in an work imbalance at runtime, with many workers staying idle. Although cilk_for also suffers from the same delayed start, workers continue performing random steals and assisting each other after completing a subset of iterations and can achieve better load balance. Enabling iteration stealing allows our CILK_PARFOR loops to edge out over cilk_for since they are able to get good load balance to counter act the delayed starts but get better locality with initial static assignments.

6.6.4 Investigating Our Overhead

The block sizes selected for the previous experiment was result of an observation of the overhead introduced in our CILK_PARFOR loops. We found that when individual iteration workloads were

small, the added call to __check_for_iteration would dominate the execution. Recall this function called by a worker while executing an assigned set of iterations and is responsible for preventing races on individual iterations.

When we run Conway's Game of Life with smaller a block size of 16x16, we spent 82% of the total time executing __check_for_iteration, prohibiting our CILK_PARFOR loops from being competitive in this case.

Using larger block sizes allows us to avoid the issues associated with this overhead, however this limits our CILK_PARFOR loop's performance on smaller applications as larger blocking factors result in less overall parallelism.

6.7 Conclusions and Future Work

PARALLEL_FOR loops offer a convenient way for programmers to quickly parallelize their applications. Many of these loop based applications can achieve perfect locality with PARALLEL_FOR loops when iterations are evenly distributed and executed on the same processor over and over. If an even distribution of iterations translates to an even work distribution, this static assignment results in good load-balance. It may be the case, however, that loop iterations have differing amounts of work and a simple static assignment strategy will be incapable of finding a load-balanced schedule.

CILKPLUS' cilk_for, on the other hand, naively achieve load balance at the cost of providing no guarantees about what executes where. As a result, their cilk_for loops cannot take advantage of the locality inherent in many loop-based applications.

Our CILK_PARFOR loops are able to statically assign iterations to workers, enabling them to exploit the locality benefits of continually executing the same iterations. On top of this, we support dynamic load-balancing by allowing workers to steal iterations after they have completed their initial assignment. We find that in applications which are not sensitive to locality, our implementation does not perform significantly worse than vanilla cilk_for. On data bound applications with regular memory access patterns, which are ideal for OPENMP's static scheduling, CILK_PARFOR is able to edge out above cilk_for, getting much closer to OPENMP's performance.

For future work, it would be ideal to implement CILK_PARFOR as a language keyword, which would help simplify the process of porting existing applications to use it. Doing so would require adding the ability to select grain sizes (the smallest schedulable unit), use arbitrary iteration types and test conditions for completion.

Additionally we would like to explore how to enable parallelism within an iteration as well as supporting dynamic discovery of iteration assignments that can be saved and replayed.

References

[1] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[2] Kunal Agrawal, Anne Benoit, Fanny Dufossé, and Yves Robert. Mapping filtering streaming applications with communication costs. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, pages 19–28, 2009.

[3] Kunal Agrawal, Anne Benoit, Fanny DufossÃl', and Yves Robert. Mapping filtering streaming applications. *Algorithmica*, pages 1–51, September 2010.

[4] Kunal Agrawal, Anne Benoit, and Yves Robert. Mapping linear workflows with computation/communication overlap. In *ICPADS '08: Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems*, pages 195–202, Washington, DC, USA, 2008. IEEE Computer Society.

[5] Kunal Agrawal and Jeremy Fineman. Brief announcement: Cache-oblivious scheduling of streaming pipelines. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, pages 79–81, 2014.

[6] Kunal Agrawal, Jeremy T. Fineman, Jordan Krage, Charles E. Leiserson, and Sivan Toledo. Cache-conscious scheduling of streaming applications. In *Proceedings of the 24th ACM Symposium on Parallelism in algorithms and architectures*, pages 236–245, 2012.

[7] Kunal Agrawal, Charles E Leiserson, and Jim Sukha. Executing task graphs using work-stealing. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[8] Lars Arge, Michael T. Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, pages 197–206, 2008.

[9] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[10] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 66. IEEE Computer Society Press, 2012.

[11] Jonathan C Beard, Peng Li, and Roger D Chamberlain. Raftlib: a c++ template library for high performance stream parallel processing. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 96–105. ACM, 2015.

[12] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. Concurrent cache-oblivious B-trees. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 228–237, July 2005.

[13] A Benoit, M. Hakem, and Y. Robert. Optimizing the latency of streaming applications under throughput and reliability constraints. In *International Conference on Parallel Processing*, pages 325–332, Sept 2009.

[14] A Benoit and Y. Robert. Complexity results for throughput and latency optimization of replicated and data-parallel workflows. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, pages 497–506, Sept 2007.

[15] Anne Benoit and Yves Robert. Mapping pipeline skeletons onto heterogeneous platforms. *Journal of Parallel and Distributed Computing*, 68(6):790–808, 2008.

[16] Guy E. Blelloch, Rezaul A. Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 501–510, 2008.

[17] S. H. Bokhari. Partitioning problems in parallel, pipeline, and distributed computing. *IEEE Trans. on Computers*, 37(1):48–57, January 1988.

[18] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[19] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[20] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. PaRSEC: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[21] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. Dague: A generic distributed dag engine for high performance computing. *Parallel Computing*, 38(1):37–51, 2012.

[22] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. on Graphics*, 23(3):777–786, August 2004.

[23] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, et al. Concurrent collections. *Scientific Programming*, 18(3-4):203–217, 2010.

[24] Edouard Bugnion, Jennifer M Anderson, Todd C Mowry, Mendel Rosenblum, and Monica S Lam. Compiler-directed page coloring for multiprocessors. In *ACM SIGPLAN Notices*, volume 31, pages 244–255. ACM, 1996.

[25] Ernie Chan, Enrique S Quintana-Orti, Gregorio Quintana-Orti, and Robert Van De Geijn. SuperMatrix out-of-order scheduling of matrix operations for smp and multi-core architectures. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 116–125. ACM, 2007.

[26] Sanjay Chatterjee, Sagnak Tasirlar, Zoran Budimlic, Vincent Cavé, Milind Chabbi, Max Grossman, Vivek Sarkar, and Yonghong Yan. Integrating asynchronous task parallelism with MPI. In *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013*, pages 712–725, 2013.

[27] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 105–115, New York, NY, USA, 2007. ACM.

[28] Trishul M. Chilimbi and Ran Shaham. Cache-conscious coallocation of hot data streams. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 252–262, 2006.

[29] Hyeong-Ah Choi and Bhagirath Narahari. Algorithms for mapping and partitioning chain structured parallel computations. In *Proceedings of the International Conference on Parallel Processing*, pages 625–628, 1991.

[30] Daniel Cordes, Andreas Heinig, Peter Marwedel, and Arindam Mallik. Automatic extraction of pipeline parallelism for embedded software using linear programming. In *Proceedings of IEEE 17th Interenational Conference on Parallel and Distributed Systems*, pages 699–706, 2011.

[31] T Cormen, C Leiserson, R Rivest, and C Stein. *Introduction to Algorithms*. MIT Press, 2009.

[32] Mark A. Franklin, Eric J. Tyson, James H. Buckley, Patrick Crowley, and John Maschmeyer. Auto-pipe and the X language: A pipeline design tool and description language. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, April 2006.

[33] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the Cilk-5 multithreaded language. In *ACM Sigplan Notices*, volume 33, pages 212–223. ACM, 1998.

[34] Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. Mining data streams: a review. *SIGMOD Rec.*, 34(2):18–26, June 2005.

[35] Thierry Gautier, Joao VF Lima, Nicolas Maillard, and Bruno Raffin. XKaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 1299–1308. IEEE, 2013.

[36] Michael I Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *ACM SIGOPS Operating Systems Review*, 40(5):151–162, 2006.

[37] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *ACM Sigplan Notices*, volume 45, pages 341–342. ACM, 2010.

[38] P. Hansen and K.-W. Lih. Improved algorithms for partitioning problems in parallel, pipelined, and distributed computing. *IEEE Trans. on Computers*, 41(6):769 –771, June 1992.

[39] Mor Harchol-Balter. The effect of heavy-tailed job size. distributions on computer system design. In *Proceedings of ASA-IMS Conference on Applications of Heavy Tailed Distributions in Economics*, 1999.

[40] Mor Harchol-Balter and Allen B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, 1997.

[41] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable stream processors. *Computer*, 36(8):54–62, August 2003.

[42] B. Khailany, W.J. Dally, S. Rixner, U.J. Kapasi, P. Mattson, J. Namkoong, J.D. Owens, B. Towles, and A. Chang. Imagine: Media processing with streams. *IEEE Micro*, pages 35–46, March/April 2001.

[43] Sanjeev Kohli. Cache aware scheduling for synchronous dataflow programs. Technical Report UCB/ERL M04/3, EECS Department, University of California, Berkeley, 2004.

[44] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 114–124, 2008.

[45] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, December 1999.

[46] Laboratory for web algorithmics. `http://law.di.unimi.it/datasets.php`.

[47] I Lee, Ting Angelina, Charles E Leiserson, Tao B Schardl, Zhunping Zhang, and Jim Sukha. On-the-fly pipeline parallelism. *ACM Transactions on Parallel Computing*, 2(3):17, 2015.

[48] Will Leland and Teunis J. Ott. Load-balancing heuristics and process behavior. In *Proceedings of the 1986 ACM SIGMETRICS Joint International Conference on Computer Performance Modelling, Measurement and Evaluation*, pages 54–69, New York, NY, USA, 1986. ACM Press.

[49] Hui Li, Sudarsan Tandri, Michael Stumm, and Kenneth C Sevcik. Locality and loop scheduling on numa multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages II–140. Citeseer, 1993.

[50] Peng Li, Kunal Agrawal, Jeremy Buhler, and Roger D. Chamberlain. Adding data parallelism to streaming pipelines for throughput optimization. In *Proceedings of the 20th International Conference on High Performance Computing*, pages 20–29, 2013.

[51] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V Kale. Optimizing data locality for fork/join programs using constrained work stealing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 857–868. IEEE Press, 2014.

[52] GQ Liu, Kim-Leng Poh, and Min Xie. Iterative list scheduling for heterogeneous computing. *Journal of Parallel and Distributed Computing*, 65(5):654–665, 2005.

[53] Ying Liu, Nithya Vijayakumar, and Beth Plale. Stream processing in data-driven computational science. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, pages 160–167, 2006.

[54] Evangelos P Markatos and Thomas J LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed systems*, 5(4):379–400, 1994.

[55] Seung-Jai Min, Costin Iancu, and Katherine Yelick. Hierarchical work stealing on manycore clusters. In *5th Conf. on Partitioned Global Address Space Prog. Models*, 2011.

[56] Arno Moonen, Marco Bekooij, Rene Van Den Berg, and Jef Van Meerbergen. Cache aware mapping of streaming applications on a multiprocessor system-on-chip. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 300–305, 2008.

[57] Frank Mueller. Compiler support for software-based cache partitioning. In *ACM Sigplan Notices*, volume 30, pages 125–133. ACM, 1995.

[58] Stephen L. Olivier, Allan K. Porterfield, Kyle B. Wheeler, and Jan F. Prins. Scheduling task parallelism on multi-socket multicore systems. In *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '11, pages 49–56, New York, NY, USA, 2011. ACM.

[59] Bjørn Olstad and Fredrik Manne. Efficient partitioning of sequences. *IEEE Transactions on Computers*, 44(11):1322–1326, 1995.

[60] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: bringing order to the web. 1999.

[61] Josep M Perez, Rosa M Badia, and Jesus Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142–151. IEEE, 2008.

[62] Ali Pınar and Cevdet Aykanat. Fast optimal load balancing algorithms for 1d partitioning. *Journal of Parallel and Distributed Computing*, 64(8), 2004.

[63] Jean-Noël Quintin and Frédéric Wagner. Hierarchical work-stealing. In *Euro-Par 2010-Parallel Processing*, pages 217–229. Springer, 2010.

[64] Moinuddin K Qureshi and Yale N Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432. IEEE Computer Society, 2006.

[65] John W. Romein, P. Chris Broekema, Ellen van Meijeren, Kjeld van der Schaaf, and Walther H. Zwart. Astronomical real-time streaming signal processing on a Blue Gene/L supercomputer. In *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 59–66, 2006.

[66] Janis Sermulins, William Thies, Rodric Rabbah, and Saman Amarasinghe. Cache aware optimization of stream programs. *ACM SIGPLAN Notices*, 40(7):115–126, 2005.

[67] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[68] Jaspal Subhlok and Gary Vondran. Optimal mapping of sequences of data parallel tasks. In *Proceedings of 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 134–143, 1995.

[69] Jaspar Subhlok and Gary Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 62–71. ACM, 1996.

[70] G Edward Suh, Larry Rudolph, and Srinivas Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[71] W. Thies, M. Karczmarek, and S.P. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, 2002.

[72] Nagavijayalakshmi Vydyanathan, Sriram Krishnamoorthy, Gerald M. Sabin, Ümit V. Çatalyürek, Tahsin M. Kurç, P. Sadayappan, and Joel H. Saltz. An integrated approach to locality-conscious processor allocation and scheduling of mixed-parallel applications. *IEEE Trans. Parallel Distrib. Syst.*, 20(8):1158–1172, 2009.

[73] Zheng Wang and Michael FP O'Boyle. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 307–318. ACM, 2010.

[74] Joseph G Wingbermuehle, Roger D Chamberlain, and Ron K Cytron. Scalapipe: A streaming application generator. In *Application Accelerators in High Performance Computing (SAAHPC), 2012 Symposium on*, pages 44–53. IEEE, 2012.

[75] Tao Yang and Apostolos Gerasoulis. PYRROS: static task scheduling and code generation for message passing multiprocessors. In *Proceedings of the 6th international conference on Supercomputing*, pages 428–437. ACM, 1992.

[76] Geore Kingsley Zipf. *Selected studies of the principle of relative frequency in language*. Cambridge, Mass. : Harvard University Press, 1932.