

**PROGRAMMING FRAMEWORKS FOR
PERFORMANCE DRIVEN SPECULATIVE
PARALLELIZATION**

A Thesis
Presented to
The Academic Faculty

by

Kaushik Ravichandran

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

Georgia Institute of Technology
December 2014

Copyright © 2014 by Kaushik Ravichandran

**PROGRAMMING FRAMEWORKS FOR
PERFORMANCE DRIVEN SPECULATIVE
PARALLELIZATION**

Approved by:

Professor Santosh Pande, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Richard Vuduc
School of Computer Science
Georgia Institute of Technology

Professor Karsten Schwan
School of Computer Science
Georgia Institute of Technology

Professor Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Professor Sudhakar Yalamanchili
School of Electrical and Computer
Engineering
Georgia Institute of Technology

Date Approved: 14 July 2014

To my family,

ACKNOWLEDGEMENTS

I would like to sincerely thank my advisor Dr. Santosh Pande for all his help, guidance, motivation and funding through my Ph.D. He has been instrumental in shaping me as a researcher. His enthusiasm in tackling challenging problems and his knack of identifying promising solutions is something I greatly admire. He has been a mentor and a guide, helping me reach my full potential. I would also like to thank him for providing excellent lab facilities, equipment and travel assistance.

I would also like to thank all the members of my thesis committee Dr. Richard Vuduc, Dr. Karsten Schwan, Dr. Hyesoon Kim and Dr. Sudhakar Yalamanchili for their comments, suggestions and feedback which helped shape this thesis. I would also like to thank Dr. Ada Gavrilovska whom I got the opportunity to work with.

I am especially grateful to my current and former colleagues in the lab. Romain helped me through the initial few years and for his mentoring I am very thankful. I would also like to thank Jaswanth, Tushar and Changhee who were always there to provide advice and Sangho for always helping out. I would also like to thank the many people in the lab including Nagesh, Shauvik, Chris, Girish, Pranith and Vincent for being great people to work with.

The last few years would not have been the same if not for the many friends outside the lab who provided the support system, making sure things never got too monotonous and making my entire experience very enjoyable.

My parents and my sister have been and continue to be wonderfully supportive providing a great deal of strength and encouragement, making all of this possible and for this I am very grateful.

TABLE OF CONTENTS

DEDICATION	iii
ACKNOWLEDGEMENTS	iv
LIST OF TABLES	ix
LIST OF FIGURES	x
SUMMARY	xiii
I INTRODUCTION	1
1.1 Controlling Degree of Speculation	3
1.2 Wasted Work and Rollbacks in Speculations	4
1.3 Guiding Speculations for Improved Performance	4
1.4 Development of Speculative Applications	5
1.5 Scalability of Speculations	6
1.6 Thesis Statement and Contributions	6
1.6.1 Contributions	6
II CONCURRENCY CONTROL FOR SPECULATION	8
2.1 Introduction	8
2.2 F2C2-STM Design	12
2.2.1 Transaction Throughput	13
2.2.2 Concurrency Fluctuations	14
2.2.3 Thread Gating	17
2.2.4 Hand-Offs	19
2.3 Evaluation	21
2.3.1 Methodology	21
2.3.2 Scalability Limited Benchmarks	24
2.3.3 Fully Scalable Benchmarks	30
2.4 Related Work	32
2.5 Summary	36

III MERGE SEMANTICS FOR SPECULATION	37
3.1 Introduction	37
3.1.1 Connected Components Problem	39
3.1.2 Speculatively Parallelizing Graph Algorithms	40
3.2 Merge Construct	42
3.2.1 STM Requirements	48
3.3 Framework API	49
3.4 Properties	52
3.5 Experimental Evaluation	53
3.5.1 Minimum Spanning Tree	54
3.5.2 Connected Components	57
3.6 Related work	59
3.7 Summary	60
IV GUIDING SPECULATIONS USING DATA STRUCTURE SE-	
MANTICS	63
4.1 Introduction	64
4.1.1 Key to parallelism: data disjointedness	65
4.1.2 Motivating example	68
4.1.3 Use-cases for $D_f(O)$	70
4.2 Expressing data-structure semantics	71
4.2.1 Properties on the data-footprint	71
4.2.2 Specification	75
4.3 Runtime usage and implementation	77
4.3.1 Programmer specifications	78
4.3.2 Low-overhead runtime	80
4.3.3 Runtime usage	82
4.4 Experimental evaluation	83
4.4.1 Greedy graph coloring	83
4.4.2 STAMP benchmarks	86

4.4.3	Scaling	88
4.5	Related work	89
4.6	Summary	91
V	DETERMINISM FOR SPECULATIVE APPLICATION DEVELOPMENT	92
5.1	Introduction	92
5.2	Determinism in Multi-Threaded Code	97
5.3	Determinism in STMs	99
5.3.1	Applicability and Limitations	99
5.3.2	STM Recap	99
5.3.3	Internal Determinism	100
5.3.4	Invariants	100
5.3.5	Non-determinism due to Memory Conflicts	101
5.4	Double Barrier Technique for STMs	102
5.5	DeSTM	104
5.5.1	Removing Barrier 1	105
5.5.2	Removing Barrier 2	107
5.5.3	Randomizing the Execution Schedule	109
5.6	Evaluation	111
5.6.1	Determinism	111
5.6.2	Performance	112
5.6.3	Scalability	114
5.6.4	Overheads	115
5.6.5	Invariant Based Bug Identification	116
5.7	Related Work	119
5.8	Summary	121
VI	EFFICIENTLY SUPPORTING DISTRIBUTED ALGORITHMIC SPECULATION	122
6.1	Introduction	122

6.1.1	Motivation	124
6.1.2	Multiverse Execution Model and Terminology	126
6.2	Multiverse Usage	128
6.2.1	Limitations	131
6.3	Multiverse Design	131
6.3.1	Code Section	133
6.3.2	Execution Stack and Context	134
6.3.3	Heap Section	138
6.3.4	Globals Section	142
6.4	Complete Execution Flow and API	142
6.4.1	API and Usage	144
6.5	Scalability	146
6.5.1	Heap Memory Accesses	146
6.5.2	Stack and Execution Context	148
6.6	Evaluation	149
6.6.1	WalkSAT	152
6.6.2	All Interval Series Problem	153
6.6.3	Costas Arrays	155
6.6.4	Perfect Square Problem	155
6.6.5	Traveling Salesman Problem (TSP)	157
6.6.6	Overheads & Scalability	159
6.7	Related Work	161
6.8	Summary	163
VII	CONCLUSION	165
7.1	Future Work	167
APPENDIX A	— PAGE ACCESS COUNTS	169
REFERENCES	171

LIST OF TABLES

1	Summary of API	50
2	Classification of STAMP Benchmarks	94
3	Overheads in Cluster A and Cluster B	160

LIST OF FIGURES

1	Execution Time, CPU Usage and Num Aborts for <i>intruder</i> Benchmark	9
2	Concurrency Control in a Feedback-driven Loop	13
3	Sample <i>cwnd</i> Fluctuations	16
4	Execution Profile for the <i>intruder</i> benchmark using F2C2-STM (32 threads)	16
5	Pseudo-code for the Thread Gate	18
6	Execution Time and CPU Usage for <i>intruder</i> Benchmark	20
7	F2C2-STM Design	21
8	Execution Time, Resource Usage and Profile for <i>intruder</i> Benchmark	25
9	Execution Time, Resource Usage and Profile for <i>yada</i> Benchmark . .	27
10	Execution Time, Resource Usage and Profile for <i>vacation</i> Benchmark	28
11	Execution Time, Resource Usage and Profile for <i>kmeans</i> Benchmark	29
12	Execution Time, Resource Usage and Profile for <i>labyrinth</i> Benchmark	31
13	Execution Time, Resource Usage and Profile for <i>genome</i> Benchmark .	32
14	Execution Time, Resource Usage and Profile for <i>ssca2</i> Benchmark . .	33
15	Pseudocode for connected components problem	39
16	Threads with different start points	40
17	Preliminary pseudocode for the MERGE function. <i>t1</i> = continuing transaction, <i>t2</i> = aborting transaction.	42
18	Transaction Schedule for T1 and T2	43
19	Code snippets	44
20	Complete pseudocode for connected components with <i>merge</i>	48
21	Pseudocode for Prim's with merge	55
22	Results of the Minimum Spanning Tree benchmark	56
23	Results of the Connected Components benchmark	62
24	Motivating example: a greedy graph coloring algorithm is shown in 24(a) and a modified version using our abstractions is shown in 24(b).	69

25	Simplified runtime algorithm to determine whether <code>currentOperation</code> can execute in parallel with the existing operations.	81
26	The Greedy Graph Coloring Benchmark	85
27	STAMP Benchmarks	88
28	Labyrinth Benchmark	90
29	Input and Output Meshes for the <i>yada</i> Benchmark	95
30	The Double Barrier Technique	98
31	The Effects of ASLR on STM Conflict Detection	102
32	The Double Barrier Technique	104
33	Effect of Removing Barrier 1	106
34	Effect of Removing Barrier 2	108
35	Comparison Between the Two Techniques	109
36	Token Passing Mechanism	110
37	Randomizing the <i>next_token_owner</i> Array	110
38	Performance Comparison Between DB Approach, DeSTM and Non-Deterministic Executions	112
39	Scalability of the DB and DeSTM Executions	114
40	Breakdown of Overheads in Deterministic Executions	116
41	Original <i>vacation</i> Code	117
42	Fixed <i>vacation</i> Code	117
43	WalkSAT Execution Time, CDF and Speedup	125
44	Basic Multiverse Speculative Execution Model	127
45	Example Multiverse Usage	129
46	Multiverse Speculative Execution Model	132
47	Virtual Address Space	136
48	Stack Switching (Multiverse Stack and Standard Stack)	137
49	Pseudo-code for Signal Handler	140
50	Detailed Execution Model	143
51	Page Server Cache Tree	147

52	Results from WalkSAT Benchmark on different datasets	151
53	Results for frb45-21-2.cnf with a larger number of cores	153
54	Results from the All Interval Series Benchmark	154
55	Results from Costas Array Benchmark	156
56	Results from the Perfect Square Benchmark	157
57	Results for TSP Benchmark for Cluster B	158
58	Multiverse Scalability	161
59	Page Access Counts	170

SUMMARY

Effectively utilizing available parallelism is becoming harder and harder as systems evolve to many-core processors with many tens of cores per chip. Automatically extracting parallelism has limitations whereas completely redesigning software using traditional parallel constructs is a daunting task that significantly jeopardizes programmer productivity. On the other hand, many studies have shown that a good amount of parallelism indeed exists in sequential software that remains untapped. How to unravel and utilize it successfully remains an open research question.

Speculation fortunately provides a potential answer to this question. Speculation provides a golden bridge for a quick expression of "potential" parallelism in a given program. While speculation at extremely fine granularities has been shown to provide good speed-ups, speculation at larger granularities has only been attempted on a very small scale due to the potentially large overheads that render it useless. The transactional construct used by STMs can be used by programmers to express speculation since it provides atomicity and isolation while writing parallel code. However, it was not designed to deal with the semantics of speculation. This thesis contends that by incorporating the semantics of speculation new solutions can be constructed and speculation can provide a powerful means to the hard problem of efficiently utilizing many-cores with very low programmer efforts.

This thesis takes a multi-faceted view of the problem of speculation through a combination of programming models, compiler analysis, scheduling and runtime systems and tackles the semantic issues that surround speculation such as determining the right degree of speculation to maximize performance, reuse of state in rollbacks, providing probabilistic guidance for minimizing conflicts, deterministic execution for

debugging and development, and providing very large scale speculations across distributed nodes.

First, we present F2C2-STM, a high performance flux-based feedback-driven concurrency control technique which automatically selects and adapts the degree of speculation in transactional applications for best performance. Second, we present the Merge framework which is capable of salvaging useful work performed during an incorrect speculation and incorporates it towards the final commit. Third, we present a framework which has the ability to leverage semantics of data structures and algorithmic properties to guide the scheduling of concurrent speculative transactions to minimize conflicts and performance loss. Fourth, we present DeSTM, a deterministic STM designed to aid the development of speculative transactional applications for repeatability without undue performance loss.

These contributions significantly enhance the use of transactional memory as a speculative idiom improving the efficiency of speculative execution as well as simplify the development process.

Finally, we focus on a performance oriented view of speculation, namely choose one of many speculative semantics, dubbed as algorithmic speculation. We present, the Multiverse framework which scales algorithmic speculation to a large distributed cluster with thousands of cores while maintaining its simplicity and efficiency.

To conclude, speculative algorithms are benefited by the contributions of this thesis due to the enhancements to the transactional and the algorithmic speculative paradigms developed in this work, laying the foundation for the development and tuning of new speculative algorithms.

CHAPTER I

INTRODUCTION

Today, the amount of parallel resources available to a programmer is continuously increasing. Leaps in hardware technology are ensuring a never ending supply of more parallel cores for use. Programming models have not kept up and it is becoming increasingly difficult to extract parallelism through traditional means. Programmers do not find designing and writing parallel code easy. Writing a parallel program (based on parallel algorithms) is tough. Writing a correct parallel program is tougher. Writing a correct and highly efficient parallel program is even tougher. Designing parallel algorithms, in the first place, to leverage large amounts of parallelism is the toughest! With pitfalls like deadlocks, livelocks, scheduling issues and race conditions it is easy to see why most programmers shy away from parallel programming.

With the increasing amounts of parallelism available at the hardware level there is a rising burden on programmers to utilize all of these resources. Speculation often leads to an easier and more practical path to adding parallelism to applications, a golden bridge for a quick expression of "potential" parallelism and brings a refreshingly orthogonal view to the problem. Speculation has long been used at the hardware level to make use of parallel resources. Common manifestations of speculation in hardware are branch prediction and prefetching. At higher levels, compilers can employ speculation in the form of Thread Level Speculation [117, 53]. These fine grain hardware and compiler mechanisms are typically not under the control of the programmer, have relatively low overheads and demonstrate modest gains at small scales. Whether speculation at larger granularities and larger scales can tap into the latent parallelism in applications is an open research question.

Software Transactional Memory (STM) systems [112, 54] have aimed at simplifying the development process of parallel programs. They provide a transactional construct which simplifies synchronization of shared data by providing atomicity and isolation. The transactional construct has now been integrated into gcc [64] and hardware support has started to appear in the latest mainstream processors [65]. The transactional construct that STMs expose can be used to express speculative parallelism. However, they were designed to mimic the style of database transactions and were not designed to deal with the semantics of speculation. This thesis contends that by incorporating the semantics of speculation new solutions can be constructed and speculation can provide a powerful means to the hard problem of efficiently utilizing many-cores with very low programmer efforts.

This thesis takes a multi-faceted view of the problem of speculation through a combination of programming models, compiler analysis, scheduling and runtime systems and tackles the semantic issues that surround speculation such as determining the right degree of speculation to maximize performance, reuse of state in rollbacks, providing probabilistic guidance for minimizing conflicts, deterministic execution for debugging and development, and providing very large scale speculations across distributed nodes.

We first focus on the transactional speculative paradigm and present several techniques which significantly improve the efficiency of speculative execution as well as simplify the development process, we then focus on the algorithmic speculative paradigm (choose one of many speculative semantics).

Transactional Speculative Paradigm Software Transactional Memory or STM systems are becoming increasingly popular as an elegant, programmer friendly solution to writing concurrent code. STMs provide the programmer with an atomic construct (called a *transaction*) which can be used to wrap accesses to shared data.

STMs speculatively run transactions in parallel and monitor their runtime execution for memory conflicts. On encountering a conflict the STM will abort and rollback one of the transactions, hence leading to a correct execution. Reads and writes in these transactions logically occur at a single instance of time; intermediate states are not visible to other successful transactions. STMs log operations during transactional execution which can then be used in case a rollback needs to be performed. Compared with locking techniques, STMs greatly simplify the development process, the conceptual understanding of multi-threaded programs and help make them more maintainable by working in harmony with existing high-level abstractions such as objects and modules. They side-step many of the issues that arise with lock based code, freeing programmers from concerns such as lock placement, deadlock, livelock, data consistency, atomicity or priority inversion. Transactions are also composable which makes them much easier to use. STMs have been found suitable for parallelizing many applications and provide performance gains with minimal programmer effort [21].

In addition to their conceptual benefits STMs are also very speculative (optimistic) in nature. The benefit of this speculative approach is increased concurrency. Threads do not need to wait to access a resource, and different threads can simultaneously modify disjoint parts of a data structure safely leading to performance gains. While, they are a great tool allowing for the expression of speculative parallelism, they were not explicitly designed with speculation in mind.

1.1 Controlling Degree of Speculation

The first issue we tackle is in automatically controlling the degree of speculation for best performance. Programmers can relatively easily write a speculative program using STMs but tuning it for best performance is not a straight-forward task. This is because, there is only a particular intrinsic level of speculation that applications can

support, often leading to significant performance degradation at higher levels of speculation. While, STMs have been found suitable for parallelizing many applications and have so far been able to leverage increasing core counts and have demonstrated sizable performance gains, with larger and larger core counts being made available, many applications are unable to harness this additional parallelism and are beginning to exhibit decreased performance at these higher core counts due to the unsustainable levels of speculation. In Chapter 2 we present F2C2-STM, a high performance flux-based feedback-driven concurrency control technique which automatically selects and adapts the degree of speculation in these applications providing significantly improved performance as well as resource utilization. Our technique also frees the programmer from having to worry about any concurrency specifications.

1.2 Wasted Work and Rollbacks in Speculations

While, controlling the degree of speculation can provide sizable performance gains with no programmer effort, alternate ways of exposing the semantic meaning of the speculation and allowing the programmer to mitigate the impact of incorrect speculations can help scale speculative algorithms. During misspeculation is all the work performed by a incorrect speculation wasted? Or is something salvageable? We address these questions by presenting the Merge framework for STMs in Chapter 3. While STMs are often used for speculation in irregular applications such as those based on graphs and trees, they do not deal with the semantics of speculation, in particular, the fact that incorrect speculations may have in fact performed some useful work. With the Merge framework we introduce *merge* semantics for speculations which recognizes this and mitigates the cost of incorrect speculation.

1.3 Guiding Speculations for Improved Performance

Providing programmers with the necessary tools to express semantics of the dynamic data footprints of speculative operations through probabilistic hints is another way

to leverage the semantics of the speculation to help decrease conflicts and increase performance. Speculation is a double edged sword. While speculation entails misspeculating sometimes, too many misspeculations are often the leading reason for ineffectiveness. Speculating aggressively and inaccurately at larger granularities is a recipe for inefficiency. We describe how speculations can be guided through simple probabilistic programmer hints and demonstrate how it can lead to significant performance improvements. We present a framework in Chapter 4 that is capable of reasoning about the *semantics* of the dynamic data footprints of speculative operations to determine their potential overlap, to guide speculations. This knowledge allows our runtime to make either a parallelization decision or throttle concurrency to improve performance (in STMs in particular).

1.4 Development of Speculative Applications

While, STMs have significantly eased the process of writing speculative code they are not a panacea for all bugs. It is still possible to make mistakes while using the transactional construct (for example missing shared state wrapping functions) or just regular implementation issues. Further, the very nature of speculation with aborts, rollbacks and retries can make the debugging process more difficult. For these reasons its is crucial to provide developers with the tools necessary to make writing speculation code easy. STMs have helped developers side-step many of the issues with parallel code, but one of the key remaining challenges which restrict programmer productivity is the non-determinism of parallel code. Non-determinism describes the phenomenon where code behaves differently during different executions despite there being no non-determinism in the inputs. We present DeSTM in Chapter 5, a deterministic STM which not only allows programmer to develop code while using a deterministic execution model but also significantly improves the performance of deterministic executions by exploiting the properties of transactional systems.

Algorithmic Speculative Paradigm While the previous model of speculation we have addressed speculated on dependencies, we now focus on a speculative model which speculates on performance and quality of result, dubbed in literature as algorithmic speculation. The algorithmic speculative paradigm allows programmers to exploit deeper algorithmic properties to extract parallelism.

1.5 Scalability of Speculations

Algorithmic (also known as high-level or function-level) speculation is a type of speculation that can be introduced by the programmer. Many important applications demonstrate variance in execution time even on the *same input*. Other applications are amenable to being solved through multiple approaches with varying execution times or different quality of results (QoRs). For these kinds of applications, programmers can use high-level speculation to speculatively run multiple instances of the (same or different) function or algorithm *in parallel* and then dynamically choose to incorporate the results from the best one. We demonstrate how the semantics of many applications allow us to extract a large amount of parallelism and present the Multiverse framework in Chapter 6 which scales such algorithmic speculation to a distributed setting while maintaining its simplicity and efficiency.

1.6 Thesis Statement and Contributions

This dissertation aims to support the following hypothesis:

Leveraging the semantic properties of speculation can tackle the issues that surround coarse grain speculative execution by improving performance and simplifying the development process, laying the foundation for the development and tuning of new speculative algorithms.

1.6.1 Contributions

To this end, this dissertation makes the following specific contributions:

- We present F2C2-STM which automatically selects and adapts the right degree of speculation in STM applications in a highly dynamic setting. It also encourages programmers to abstain from using any concurrency specification.
- We present the Merge framework which provides an elegant way of parallelizing connected component discovery in a large graphs. It is capable of leveraging currently useful state towards the final commit, mitigating the cost of incorrect speculations and rollbacks.
- We present a framework enabling the expression of semantics of data structures and their algorithmic properties which can be used to guide speculations through probabilistic conflict estimates.
- We present DeSTM which introduces determinism into the development process of speculative applications, leading to repeatability of execution and significantly easing the development process without undue performance loss.
- We present the Multiverse programming model which allows programmers to introduce scalable distributed algorithmic speculation efficiently and with negligible effort for difficult to parallelize problems or to auto select tuning parameters.

CHAPTER II

CONCURRENCY CONTROL FOR SPECULATION

2.1 Introduction

The first issue we tackle is in automatically controlling the degree of speculation in STMs for best performance. Compared with locking techniques, the speculative constructs (transactions) that STMs provide greatly simplify the conceptual understanding of multi-threaded programs, are composable and side-step many of the issues that arise with lock based code, freeing programmers from concerns such as lock placement, deadlock, livelock, data consistency, atomicity or priority inversion.

In addition to their conceptual benefits STMs are also very speculative (optimistic) in nature. The benefit of this speculative approach is increased concurrency. Threads do not need to wait to access a resource, and different threads can simultaneously modify disjoint parts of a data structure safely leading to performance gains.

STMs have been found suitable for parallelizing many applications and provide performance gains with minimal programmer effort [21]. STMs have so far been able to leverage increasing core counts and have demonstrated sizable performance gains. However, with larger and larger core counts being made available, many applications are unable to harness this additional parallelism and are beginning to exhibit decreased performance at these higher core counts. Take for example the *intruder* benchmark from the STAMP STM suite. Figure 1(a) shows the execution time with an increasing number of cores ¹ (each core has one thread running on it).

We can clearly see that the execution time decreases up until 5 cores and then

¹Configuration: Percentage of attacks: 10% Max number of packets per stream:128 Total number of streams: 262144

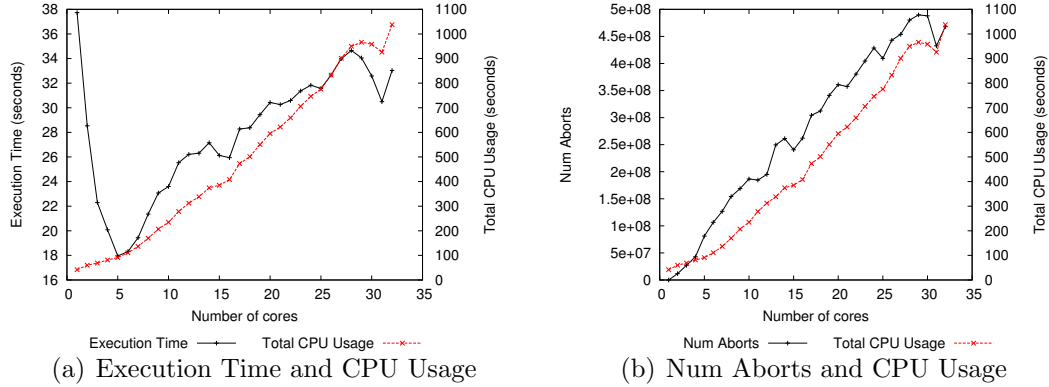


Figure 1: Execution Time, CPU Usage and Num Aborts for *intruder* Benchmark

starts increasing again. In fact the execution time when running on 32 cores is almost twice that of running on 5 cores and is almost equal to the time it takes to run on 1 core! While utilizing more cores (to run more parallel threads) should theoretically decrease the execution time, a larger number of threads also induce a larger number of conflicts (or aborts) amongst themselves (as shown in Figure 1(b)) due to higher levels of speculation (or optimism). Many applications can inherently only support a certain level of speculative executions before these executions start excessively interfering with each other resulting in conflicts and performance loss. These conflicts are essentially just completely wasted work, incur high overheads, increase resource utilization and can potentially slow down execution significantly. Figure 1(a) shows resource utilization with an increasing number of cores. We define resource utilization as the total CPU usage across all cores for an execution (CPU usage is obtained by adding the system and user components from the *time* linux command). We observe that resource utilization increases almost linearly with the number of cores (resource utilization also loosely correlates to the number of aborts, Figure 1(b)).

Overall, using a larger number of cores not only increases the execution time but also results in much higher resource utilization, resulting in an extremely detrimental effect on overall performance. As the number of cores available in a single machine

is steadily on the rise, a larger number of STM applications are beginning to exhibit these characteristics. We refer to these STM applications as *scalability limited* applications and they are the focus of this chapter. STM applications which do not exhibit these characteristics and can utilize all the cores available on a machine are referred to as *fully scalable* applications.

Traditionally, the *concurrency specification* (number of threads created) for an application has always been set at the number of cores available to it (oversubscription typically does not help [84, 61]). While for *fully scalable* STM applications such a specification allows for maximal use of resources, it clearly leads to sub-optimal performance for *scalability limited* STM applications. The degree of speculation is directly controlled by the concurrency level. Choosing the right concurrency specification for such applications is not a straight forward choice. It depends on numerous factors including transaction size, rollback overheads, transaction read/write-set sizes, logging overheads and hardware specifications. It can also vary based on the input data set that the application uses. For example, the optimal concurrency specification for the example dataset used in Figure 1 is 5 while the optimal specification for a different dataset is 7 (see dataset used in Section 2.3). Thus, rendering any statically fixed concurrency specification by the programmer useless. In fact, applications also exhibit *varying concurrency levels in a single execution*, which cannot be exploited by any static specification. *Concurrency control*, the act of controlling the number of threads that are allowed to run at any given time, can be used to remedy this problem.

In this chapter we present a dynamic concurrency control technique that can automatically limit the concurrency in *scalability limited* STM applications in an attempt to reduce execution time *and* improve resource utilization. Concurrency control also completely relieves the programmer or user from having to indicate any concurrency

specification at all, since the right specification is dynamically discovered. For applications which use standard threading APIs to manage concurrency our technique requires no modifications to application code and requires no offline phases.

Our dynamic concurrency control technique improves performance and resource utilization significantly for *scalability limited* applications at higher core counts. We use ideas borrowed from TCP’s network congestion control algorithm and use self-induced concurrency fluctuations to dynamically monitor and match varying concurrency levels in applications while minimizing global synchronization. Our technique is capable of fully recovering the performance of the best statically chosen concurrency specification (as chosen by an oracle) for several applications. Some applications even demonstrate varying levels of concurrency within a single run itself. Since our technique can dynamically modify the concurrency in an application at runtime, it is able to adapt to these concurrency variations and as a result in some cases we actually observe better performance than any statically chosen concurrency specification (even an oracle chosen specification). We also empirically demonstrate how our approach imposes minimal overheads on real world applications which are *fully scalable*.

We built our technique, F2C2-STM (Flux-based Feedback-driven Concurrency Control STM) on top of TinySTM [42]. F2C2-STM is able to effectively control concurrency for real world applications from the STAMP benchmark suite [21]. When compared with other state of the art concurrency control techniques F2C2-STM exhibits significantly improved performance and resource utilization.

The remainder of this chapter is organized as follows. In Section 2.2 we discuss our technique and its design. In Section 2.3 we present a comprehensive evaluation of our technique. In Section 2.4 we discuss related work.

2.2 F2C2-STM Design

F2C2-STM is built on top of TinySTM, a state of the art STM infrastructure frequently used in research. TinySTM has no concurrency control and requires a *static concurrency specification* typically taken from the command line. This specifies the number of threads that are created and that run throughout the lifetime of the application. As we saw in Section 2.1 the ideal level of concurrency for an application may be different from the *static concurrency specification*. This ideal level of concurrency may in fact vary through the execution as well. For example, the ideal level may start at 3 threads and then gradually increase to 12 by the end of the execution. We refer to this *varying ideal level* of concurrency in an application at a given point of time as the *inherent concurrency level*. F2C2-STM tries to control the number of threads that are running at any given time, i.e. the *dynamic concurrency level* in an attempt to match the *inherent concurrency level*.

F2C2-STM determines the right *dynamic concurrency level* by frequently inducing fluctuations in the current concurrency level at runtime and monitoring the resultant performance difference in a feedback driven loop as shown in Figure 2. It uses a light-weight thread gating mechanism to control the number of threads running and uses certain techniques to minimize global synchronization overheads. This allows it to quickly discover and adapt to the *inherent concurrency levels* in an application. Unlike TinySTM which requires a *static concurrency specification* on the command line, F2C2-STM encourages programmers to abstain from specifying any such information. If F2C2-STM is used without any concurrency specification, by default it launches as many threads as there are cores. This allows F2C2-STM to use as many or as few threads as it sees fit to maximize performance while minimizing resource utilization. If F2C2-STM is used with a concurrency specification on the command line, it treats this specification as the *maximum concurrency specification*, and creates only that many threads and performs concurrency control only within these, resulting in an

upper bound on the number of occupied cores.

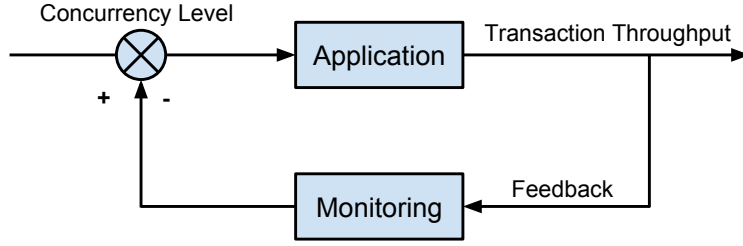


Figure 2: Concurrency Control in a Feedback-driven Loop

2.2.1 Transaction Throughput

F2C2-STM uses committed transaction throughput as the performance metric that it tries to optimize in the feedback loop. Note that when we refer to transaction throughput, we are referring only to committed transaction throughput. The dynamic transaction throughput during execution is defined as the total number of transactions that commit per unit time interval (tpi : Transactions per Interval). $tpi_T = \sum_{i=1}^n tpi_i$ where tpi_T is the total tpi for the entire application with n threads and tpi_i is the tpi of the i^{th} thread. tpi_i s are computed using simple per thread commit counters whenever required.

While, tpi_T is the ideal metric to maximize, computing tpi_T is fairly expensive as it will involve some sort of global synchronization (for example, a lock protected global counter), imposing a severe burden on execution time. *Minimizing any type of frequent global communication is key to achieving high performance concurrency control.* Hence, F2C2-STM designates one of the threads as a *head* thread and attempts to maximize tpi_{head} , using it as a proxy for tpi_T . If there is throughput loss in the system due to excessive conflicts we can assume that it effects all threads equally and that tpi_{head} will reflect tpi_T . Indeed, this assumes that there is more or less an even distribution of transactional work between all threads. All the real world benchmarks we evaluated exhibited this characteristic. Further, this necessitates the ability for

head to change in some situations, this is discussed in Section 2.2.4.

In some applications, however, where tpi_i is very low, we can in fact switch over to a global counter and maximize tpi_T itself. This is because, a low tpi_i implies fewer global counter accesses resulting in acceptable synchronization overheads. F2C2-STM automatically switches between these two modes based on the values of tpi_i . In the STAMP suite this situation occurs only in one benchmark (*labyrinth*).

Note that we are not just trying to minimize the number of conflicts. A lower number of conflicts does not necessarily translate into better performance. For example, take the extreme situation where we have only 1 (or 2) threads. We will have 0 (or few) conflicts while performance suffers due to low concurrency (as can be seen in Figure 1). Optimizing for throughput gives us a cleaner, better signal.

2.2.2 Concurrency Fluctuations

F2C2-STM essentially tries to perform an efficient search of the space of available concurrency options to find the one which maximizes transaction throughput. It uses ideas inspired by TCP’s network congestion control algorithm. To recap, TCP uses several algorithms to control congestion in a network and uses a congestion window to throttle the amount of outstanding data in the network. Two of TCP’s algorithms are of interest to us. Slow-start and Congestion Avoidance. Slow-start, contrary to an intuitive interpretation of its name, actually increases the window in an exponential manner until a timeout occurs to quickly find an approximate size for its congestion window. Subsequently, when congestion is reached, the algorithm enters a new state called congestion avoidance where the congestion window is additively increased by one unit every round trip time.

F2C2-STM uses similar ideas. It maintains a *concurrency window* ($cwnd$) which controls the number of threads that may be active at any given time. To determine the right size of this window, it uses an initial exponential search phase which we refer

to as the *Coarse Grain Search (CGS) Phase* and a subsequent linear search phase which we call the *Fine Grain Search (FGS) Phase*. Transaction throughput (*tpi*) is closely monitored between each modification to the *cwnd* in these phases by the *head* thread.

Modifications which result in better *tpi* result in a + signal while changes which worsen the *tpi* result in a – signal being used as input to compute the new *cwnd*. Modifications to *cwnd* and *tpi* monitoring are only performed once every 10ms and only by the *head* thread (during calls into the STM runtime), ensuring that the overhead has negligible impact on performance while also ensuring adaptations can be performed quickly enough. A linux timer is used to measure time intervals. Algorithm 1 briefly describes the algorithm.

Coarse Grain Search (CGS) Phase In the CGS phase an exponential search (in powers of 2) of the concurrency space is performed. CGS initially sets *cwnd* to 2, exponentially increasing it in each iteration and monitoring performance. A + signal continues the exponential search, whereas a – signal moves the algorithm in to the FGS phase after dividing the current *cwnd* by 2.

Fine Grain Search (FGS) Phase In the FGS phase a linear search (increments/decrements of 1) of the concurrency space is performed. In this phase, the algorithm will be in either *INC_MODE* or *DEC_MODE*. The first iteration of this phase starts in *INC_MODE*. If the previous change resulted in a + signal, *INC_MODE* increments *cwnd* by 1 and *DEC_MODE* decrements *cwnd* by 1. If the previous change resulted in a – signal, *INC_MODE* decrements *cwnd* by 1 and *DEC_MODE* increments *cwnd* by 1. A – signal also results in the mode toggling for the next iteration.

Fluctuations Note, that in each of these phases the value of *cwnd* will almost *always* be changing in each iteration. Even in the FGS phase a + signal will either increase or decrease *cwnd* based on the current mode (*INC_MODE/DEC_MODE*). If for some reason, the value of *cwnd* can not change for some time, say, because the

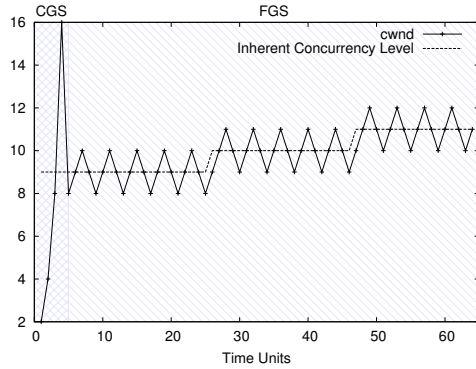


Figure 3: Sample *cwnd* Fluctuations

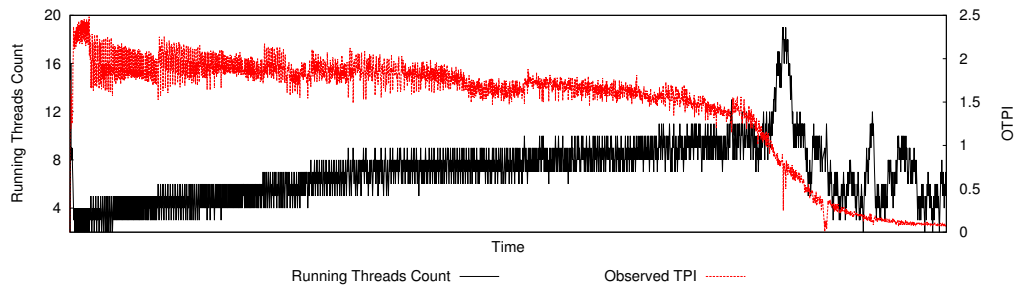


Figure 4: Execution Profile for the *intruder* benchmark using F2C2-STM (32 threads)

extreme value has been reached in that direction of the search (1 or maximum level of concurrency), the mode (*INC_MODE/DEC_MODE*) is inverted to force *cwnd* to move in the opposite direction (this is not shown in the pseudo-code in Algorithm 1). If throughput reduces in that direction, the generated $-$ signal for *cwnd* computation will reverse the direction again in the next iteration. Indeed, in the degenerate case of only 1 thread, no fluctuations can be induced. This technique of constantly changing *cwnd* allows F2C2-STM to continuously adapt to any changes in concurrency levels. *cwnd* controls the number of threads that are allowed to run at any given time. In the “steady-state” the value of *cwnd* will oscillate between an appropriate value. These oscillations ensure rapid adaptation to changes in the inherent concurrency level as shown in Figure 3.

Figure 4 is an execution profile and illustrates how these fluctuations manifest themselves in a real world benchmark (*intruder*) launched with 32 threads. The

black line shows the fluctuating dynamic concurrency level (wnd) as it adapts to the varying inherent concurrency level. The number of threads that are allowed to run start low (around 3) and then gradually increases during execution. The red line depicts the transaction throughput per interval (tpi) that is being measured dynamically. We refer to this as the *observed tpi (OTPI)*.

Note that the tpi is a function of the number of threads currently running *and* the characteristics of the benchmark. For example, we observe that towards the end of the execution in Figure 4 the tpi decreases, this is inherent in the nature of the benchmark (baseline exhibits this characteristic as well) and is due to the self-balancing tree structure used. F2C2-STM tries to continuously modify wnd in an effort to obtain the maximum possible tpi .

Algorithm 1 Algorithm to compute new wnd used by F2C2-STM (Pseudo-code)

```

1: procedure NEW_CWND(SIGNAL: computed from previous iteration)
2:   if PHASE == CGS then
3:     if SIGNAL == "+" then
4:        $wnd \leftarrow wnd * 2$ 
5:     else if SIGNAL == "-" then
6:        $wnd \leftarrow wnd / 2$ 
7:       PHASE ← FGS
8:     return
9:   if PHASE == FGS then
10:    if SIGNAL == "+" then
11:      if FGS_MODE == INC_MODE then
12:         $wnd \leftarrow wnd + 1$ 
13:      else if FGS_MODE == DEC_MODE then
14:         $wnd \leftarrow wnd - 1$ 
15:      else if SIGNAL == "-" then
16:        if FGS_MODE == INC_MODE then
17:           $wnd \leftarrow wnd - 1$ 
18:          FGS_MODE = DEC_MODE
19:        else if FGS_MODE == DEC_MODE then
20:           $wnd \leftarrow wnd + 1$ 
21:          FGS_MODE = INC_MODE
22:    return

```

2.2.3 Thread Gating

F2C2-STM provides a light-weight thread gating mechanism which allows the runtime to perform frequent fluctuations in an effort to adapt to the inherent concurrency level.

As discussed previously, one of the running threads is the designated *head* thread

and is in charge of monitoring performance and computing the new *cwnd*. In addition, the *head* thread also *gates* and *ungates* other threads. If *cwnd* has increased the required number of threads are ungated else if *cwnd* has decreased the required number of threads are gated by the *head* thread.

Gates Each thread has an individual gate used to control it. A thread encounters the gate when it makes a call to begin an STM transaction. An unlocked gate allows the thread to pass through unhindered while a locked gate stops the threads until it is unlocked. The overhead of the complete gating mechanism needs to be kept as low as possible. This not only ensures that better performance can be obtained while controlling concurrency but importantly also ensures that the overhead imposed by such a mechanism on *fully scalable* applications is kept to a minimum. There are two code paths that a thread can take through the gate, the *opened gate* path and the *closed gate* path. While low overhead on both of these paths is preferred, minimizing the overhead on the *opened gate* path is more important to ensure minimal overheads while a thread is clear to run. A higher overhead can be sustained on the *closed gate* path since the thread is to wait until the gate is opened in any case. In the *closed gate* path the thread waits on a semaphore. Pseudo-code for the gating mechanism is shown in Figure 5.

```

1 void gate() {
2     // 'gated' and 'gateSemaphore' are padded to use separate cache lines.
3     if (gated[thread_id] == 1) { // Synchronized.
4         sem_wait(gateSemaphore[thread_id]);
5     }
6 }

```

Figure 5: Pseudo-code for the Thread Gate

Opened Gate In the case of an *opened gate* the thread simply has to read a value (which would typically be located in the cache), perform a check on it and continue its execution. Each thread owns its own *gated[thread_id]* variable. While these are variables in an array, they are padded to put them on different cache lines ensuring

that there is no false sharing. The *head* thread is in charge of changing the value of the gate. Accesses to *gated* variables are synchronized (low overhead due to infrequent changes, see execution profiles in Section 2.3). Note that semaphore accesses which are expensive do not occur on this path.

Closed Gate In the *closed gate* path the thread has observed that it is gated and hence simply waits on a semaphore. Again, these semaphore variables are all on different cache lines to ensure no false sharing. If the *head* thread needs to ungate the thread it will unlock the gate and *sem_post* on the thread’s semaphore to wake it up.

Thread Waiting Mechanism We evaluated two possible ways to implement the waiting mechanism (line 4 in Figure 5). In the first mechanism we used a busy wait loop which kept checking a flag controlled by the *head* thread. In the second mechanism (shown in Figure 5) we used a semaphore to put the thread to sleep until it is awoken by the *head* thread. Figure 6 shows the execution time and CPU usage observed in the *intruder* benchmark using these two approaches. Intuitively one would expect the busy wait approach to provide faster response times.

We see from Figure 6 that the semaphore approach provides comparable performance to busy wait, but is just slightly slower (less than 3% at higher core counts). However, the difference in CPU utilization using semaphores is dramatic (on an average more than a 60% improvement in usage at higher core counts). To ensure that CPU utilization also benefits from concurrency control we adopted semaphores despite the slight loss in execution time.

2.2.4 Hand-Offs

Hand-Offs occur when a thread is no longer available to run. For example, a thread may complete its execution and may be terminated. Alternatively, a thread may go to sleep at a barrier waiting for other threads to arrive or it may sleep while waiting

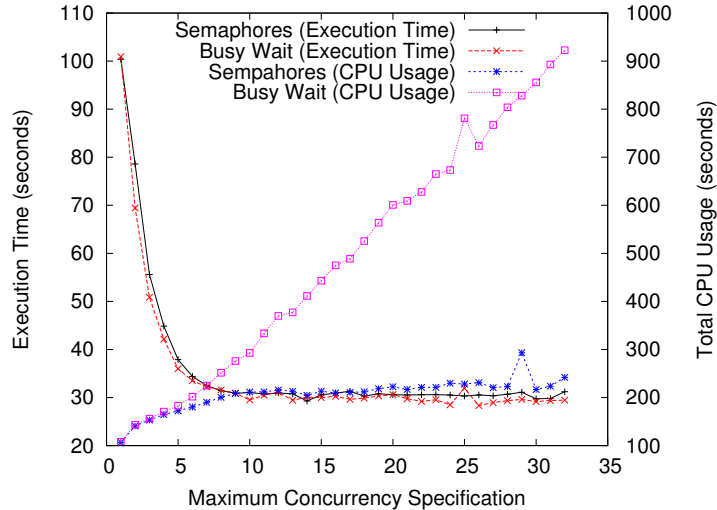


Figure 6: Execution Time and CPU Usage for *intruder* Benchmark

to communicate with another thread. These operations are typically performed by calling into the underlying threading library. F2C2-STM provides wrappers for these functions which automatically call the *handoff* routine.

The *handoff* routine ungates a gated thread (that is waiting to run) and allows it to execute. Recall that F2C2-STM maintains a *cwnd* which controls the number of threads that are allowed to run at any given point of time. Hence, this *handoff* routine gives every thread the opportunity to run in a controlled fashion.

Hand-Offs with the head thread The *head* thread is in charge of controlling the concurrency. In the situation that the *head* thread calls the *handoff* routine its responsibilities are transferred to another thread automatically during the call. Hence, in addition to allowing another thread to run the system automatically chooses a thread to mark as the new *head* thread and transfers responsibilities and lock ownership.

As long as the programmer uses the functions provided by the threading library these hand-offs happen transparently to the programmer. Alternatively, the runtime exposes a *handoff* function which can be invoked by the programmer. We found no situations in the STAMP benchmark suite where this function needed to be manually

inserted. All benchmarks used the standard threading libraries and hence hand-offs were transparently performed without programmer intervention.

Figure 7 describes the overall design of the system.

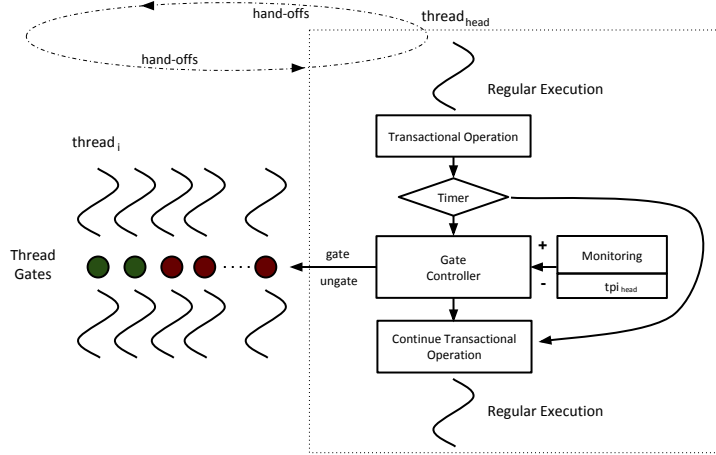


Figure 7: F2C2-STM Design

2.3 Evaluation

In this section we present a comprehensive evaluation of F2C2-STM. Our prototype implementation of F2C2-STM is built on top of TinySTM [42].

2.3.1 Methodology

We demonstrate using F2C2-STM the utility of concurrency control on real world benchmarks from the STAMP suite [21]. We demonstrate how it can recover performance in situations where the concurrency is specified incorrectly and how it can relieve the programmer from having to specify it in the first place. We demonstrate how it can even outperform an oracle chosen specification for some applications. Further, we compare performance with 2 state of the art approaches to concurrency control and scheduling.

- **ATS (Adaptive Transaction Scheduling)** [131], an interesting technique which performs concurrency control by using the notion of *contention intensity*

(*CI*) to determine when to serialize transactions. Transactions are serialized through a queuing system which results in concurrency control based on the current *CI*. We obtained the implementation from [109]. *ATS*, however, requires a sensitivity analysis to determine the right value for α , the aging factor for *CI* (as described in [131]) for best performance. Our independent sensitivity analysis resulted in the same value of α that was obtained in the original distribution.

- **Shrink STM** [41] is a novel mechanism to schedule transactions by leveraging predictions of transactional read/write sets. It identifies possible conflicts with currently running transactions using these predictions and may decide to serialize these transactions based on current contention. We obtained the implementation from [109].
- **TinySTM** [42] is a state of the art STM infrastructure frequently used in research. This is the baseline STM infrastructure. It has no concurrency control and is configured with write-back encounter time locking. *F2C2-STM*, *ATS* and *Shrink STM* are built on top of the same configuration of *TinySTM*. The implementation can be obtained from [120].

Statistics We report results of the *execution time* and *resource utilization* of the *STAMP* benchmark suite [21] on each of these systems. We also report the *execution profile* and *overheads* of *F2C2-STM* for each of these benchmarks (when no concurrency is specified). The execution time each benchmark reports in *STAMP* excludes time spent in initialization and verification. The resource utilization is defined as the aggregate CPU utilization across all cores during the entire execution. It is obtained by adding the system and user components from the *time* linux command. The execution profile for each benchmark shows the number of threads that *F2C2-STM* allowed to run and the observed transactional throughput, *OTPI* (as discussed in

Section 2.2.2) during a sample run. These profiles provide very interesting views into the dynamic concurrency levels in applications. The overhead imposed by the F2C2-STM runtime is defined as the total time spent by the *head* thread in monitoring and controlling concurrency (note, this includes time spent by every *head* thread during execution of and after hand-offs).

Oracle We also compare F2C2-STM against an oracle which can perfectly set the static concurrency specification on the TinySTM baseline. The oracle scheme simply selects the lowest observed execution time from all possible static concurrency specifications for TinySTM and the resource utilization at this specification. While this scheme is impractical in practice, we use it for comparison purposes.

We also show the serial execution time for each benchmark. The serial execution uses no threads and does not link with any STM library.

Using more threads than the available number of cores (oversubscribing or overloading) is not recommend for STM applications [84, 61] since it does not provide a performance benefit (we independently verified this as well).

All experiments were performed on a 32 core machine, with 4 Intel Xeon CPUs (X7560) at 2.27GHz (with 8 cores each) for a total of up to 32 concurrent threads. The machine was running Ubuntu 10.04 LTS (Kernel Version: 2.6.36). All experiments were compiled on gcc 4.4.3 with the O3 optimization level. All results are averaged over 5 runs.

No concurrency specification Recall, that F2C2-STM encourages programmers to abstain from specifying any concurrency specification and attempts to recover the best possible performance. To evaluate this situation, the graphs which report execution time and resource utilization display a gray vertical bar around the 32 thread mark. If a static concurrency specification has not been specified, an application typically creates as many threads as there are cores. This bar highlights that configuration reflecting how the system would perform.

We were able to classify the benchmarks in the STAMP suite as either *scalability limited* or *fully scalable* (as discussed in Section 2.1). F2C2-STM is designed for *scalability limited* applications and hence we focus on these applications for our evaluation. *Fully scalable* applications typically do not benefit from any sort of concurrency control since they are able to exploit the maximum amount of parallelism that is available on a machine. However, with ever increasing core counts on processors, these applications will also begin to exhibit limited scalability in the future at which point they will also benefit from concurrency control. We empirically demonstrate that F2C2-STM exhibits acceptably low overheads when used on these *fully scalable* applications.

2.3.2 Scalability Limited Benchmarks

These are benchmarks where the *inherent level of concurrency* is less than the maximum number of cores (32).

intruder emulates a NIDS (Signature-based network intrusion detection system)². It scans network packets for matches against a known set of intrusion signatures. Network packets are processed in parallel and go through three phases: capture, reassembly, and detection. This benchmark exhibits high contention making concurrency control challenging.

Discussion The results we obtained are plotted in Figure 8. The oracle chosen concurrency specification is 7. We see that F2C2-STM is able to successfully control the concurrency and to recover the execution time of the oracle scheme. In fact, it is able to out perform the oracle chosen specification itself consistently by about 9%. The execution profile shows that there is a varying inherent concurrency level for this benchmark. F2C2-STM is able to adapt to these variations and hence out performs the oracle. In addition to recovering execution performance it also uses

²Configuration: Percentage attacks: 20%, Maximum number of packets per stream: 256 and the total number of streams: 262025

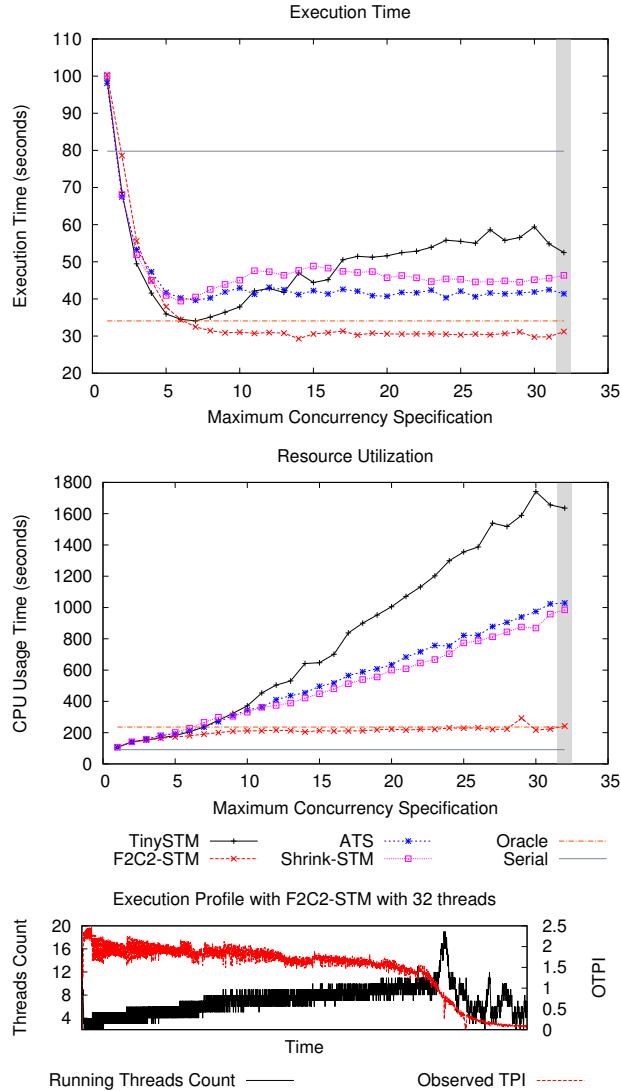


Figure 8: Execution Time, Resource Usage and Profile for *intruder* Benchmark

minimal resources compared to the other schemes. While, at lower core counts F2C2-STM has higher overheads F2C2-STM is able to significantly out perform the other versions including the baseline at higher core counts (40.5% better than TinySTM, 24.6% better than ATS and 32.6% better than Shrink STM at 32 cores). While, ATS and Shrink STM are able to control the concurrency and decrease the performance loss compared to the TinySTM baseline at higher core counts, they are unable to recover the performance of the best statically chosen specification. The overhead incurred by F2C2-STM is minimal at 0.025s. In addition, the gray bar shows that

F2C2-STM performs well in situations where no concurrency is specified.

yada (Yet Another Delaunay Application) benchmark implements Ruppert’s algorithm for Delaunay mesh refinement ³. In each iteration of the algorithm, a skinny triangle is removed from the work queue, its re-triangulation is performed on the mesh, and any new skinny triangles that result from the re-triangulation are added to the work queue for further processing. The oracle chosen concurrency specification for this benchmark is 5. The overhead incurred by F2C2-STM is 0.233s. The results we obtained are plotted in Figure 9.

vacation implements an online transaction processing system emulating a travel reservation system ⁴. The oracle chosen concurrency specification for this benchmark is 10. The overhead incurred by F2C2-STM is 0.057s. The results we obtained are plotted in Figure 10.

kmeans implements a clustering algorithm which groups objects in an n -dimensional space into k clusters ⁵. This algorithm is commonly used to partition data items into related subsets. The oracle chosen concurrency specification for this benchmark is 24. The overhead incurred by F2C2-STM is 0.15s. The results we obtained are plotted in Figure 11.

Discussion We present a condensed discussion for these *scalability limited* benchmarks. The oracle chosen specification varies from 5 to 24. The *yada* results are very similar to that of *intruder*, with F2C2-STM performing better than other schemes. It is also able to out perform the oracle scheme (by more than 10%). Similar trends are observed in the other benchmarks as well. F2C2-STM has higher overheads at lower counts but strongly outperforms the other versions including the base line (excluding oracle) at higher core counts (averaging 27.2% better than TinySTM, 40.5% better

³Configuration: Angle constraint: 15, File prefix: inputs/ttimeu1000000.2

⁴Configuration: Num queries per task: 4, % relations queried: 90%, Num possible relations: 65536, % user tasks: 98%, Num tasks: 4194304

⁵Max cluster: 40, Min clusters: 40, Threshold: 0.00001, File: random-n32768-d512-c16.txt

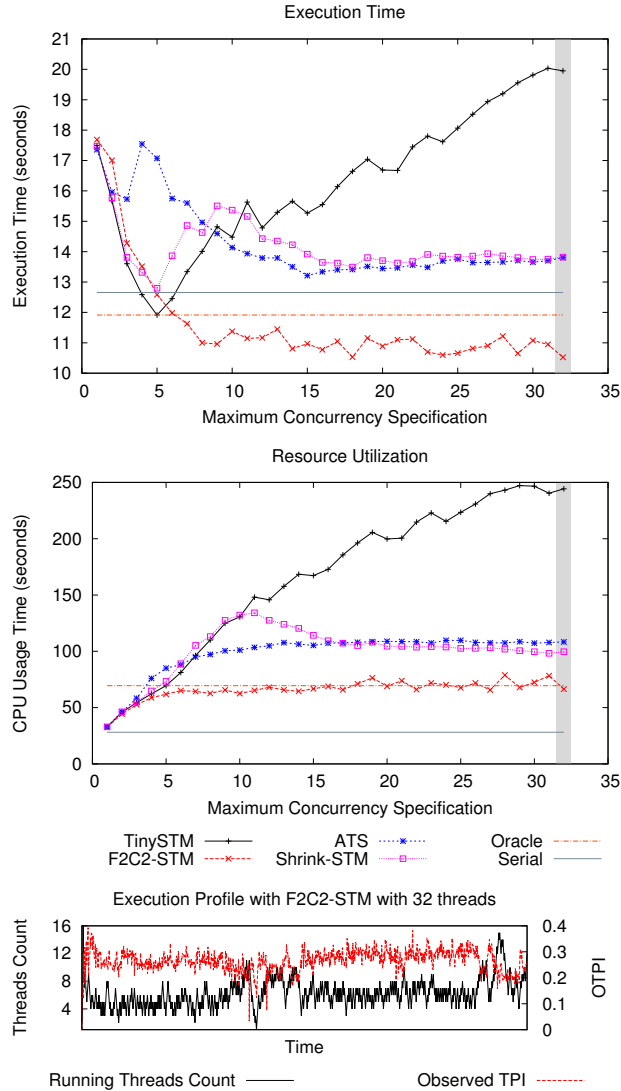


Figure 9: Execution Time, Resource Usage and Profile for *yada* Benchmark

than ATS and 44% better than Shrink STM at 32 cores). Improvements in execution time over the baseline are significant (up to 50%) with correspondingly significant improvements in resource utilization as well (up to 85%).

Overall, while ATS and Shrink STM are able to control the concurrency to a certain extent and decrease the performance loss compared to the TinySTM baseline at higher core counts they are unable to recover the best performance like F2C2-STM. For the *kmeans* benchmark, the oracle performance is in fact very close to the baseline TinySTM performance. While, F2C2-STM is unable to match or outperform

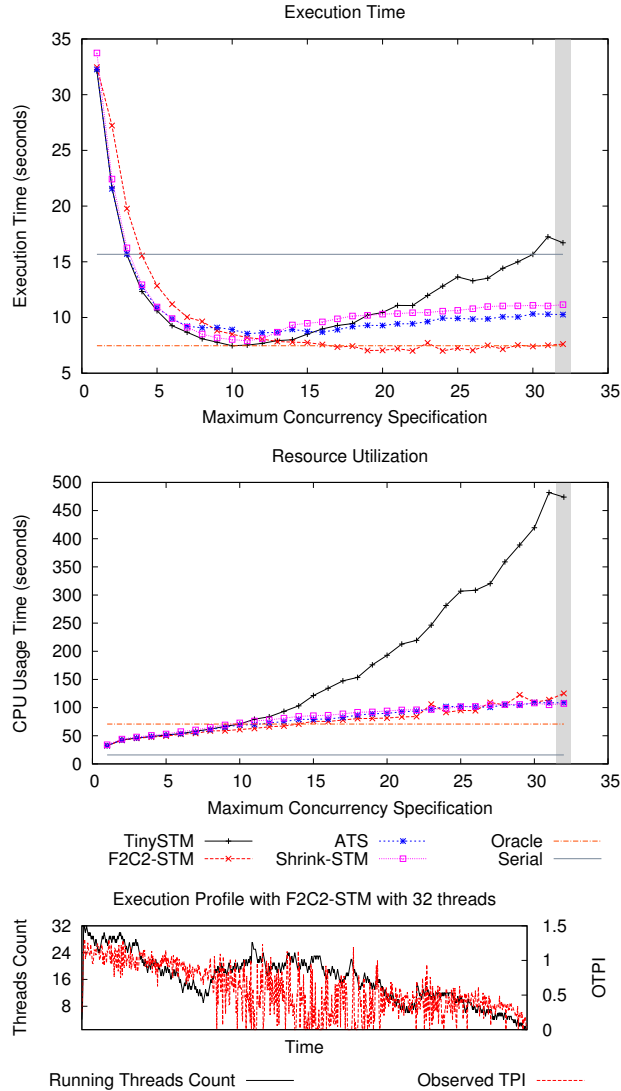


Figure 10: Execution Time, Resource Usage and Profile for *vacation* Benchmark

the baseline in this case, its performance is comparable to the baseline. ATS and Shrink STM, on the other hand, degrade in performance in this case compared to the baseline. This is due to the very short transaction sizes that *kmeans* uses which induces higher overheads for concurrency control mechanisms. It also has frequent barrier synchronizations which bring throughput down to 0 (see execution profile) (F2C2-STM quickly periodically recovers these in the CGS phase).

Qualitatively, F2C2-STM uses a more direct metric, the transaction throughput

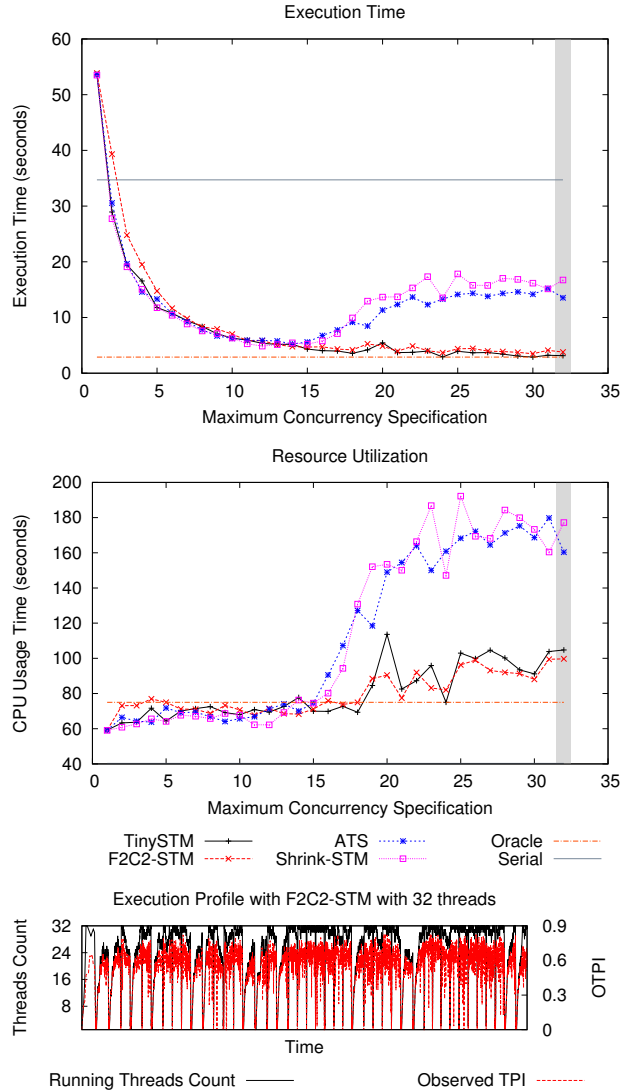


Figure 11: Execution Time, Resource Usage and Profile for *kmeans* Benchmark

(of the *head* thread) to optimize performance. Both, ATS and Shrink STM use contention to decide concurrency levels. Contention levels sometimes do not directly correlate to performance. Also, using empirically set thresholds for contention levels does not lend itself to new applications easily. Optimizing for throughput gives us a better signal. F2C2-STM’s approach of maximizing transaction throughput (tpi_{head}) by monitoring and fluctuating concurrency in a feedback loop, minimizing global synchronization as well as its light-weight gating mechanism which provides independent control over all threads (with minimal overheads) and hand-offs is able to effectively

recover execution performance in these *scalability limited* benchmarks all with improved resource utilization. Further, it is able to provide these performance benefits without any concurrency specification as well.

2.3.3 Fully Scalable Benchmarks

These are benchmarks which can use all the available parallelism and as such cannot benefit from concurrency control. Concurrency control techniques should ideally exhibit performance similar to the baseline on these benchmarks.

labyrinth implements a variant of Lee’s algorithm which is a path finding algorithm in a maze ⁶. The oracle chosen concurrency specification for this benchmark is 31. The overhead incurred by F2C2-STM is 0.044s. Note that only for this benchmark, due to extremely low t_{pi} , F2C2-STM is able to actually optimize for t_{pi_T} instead of $t_{pi_{head}}$ (as discussed in Section 2.2.1). The results we obtained are plotted in Figure 12.

genome performs genome assembly ⁷. The oracle chosen concurrency specification for this benchmark is 32. F2C2-STM overhead is 0.025s. The results we obtained are plotted in Figure 13.

ssca2 (Scalable Synthetic Compact Applications 2) ⁸ is comprised of four kernels that operate on a large, directed, weighted multi-graph. The oracle chosen concurrency specification for this benchmark is 29 (there was much variance for this benchmark, but the trend pointed towards it being fully scalable). F2C2-STM overhead is 0.432s. The results we obtained are plotted in Figure 14.

Discussion For these benchmarks concurrency control cannot improve performance. We observe that for these benchmarks, all three mechanisms F2C2-STM, ATS and Shrink STM provide comparable performance to the baseline which scales

⁶Configuration: File: random-x512-y512-z7-n512.txt

⁷Configuration: Gene length: 16384, Segment length: 64, Num segments: 16777216

⁸Configuration: Problem scale: 20, Probability of inter-clique: 1.0, Probability unidirectional: 1.0, Max path length: 3, Max parallel edges: 3

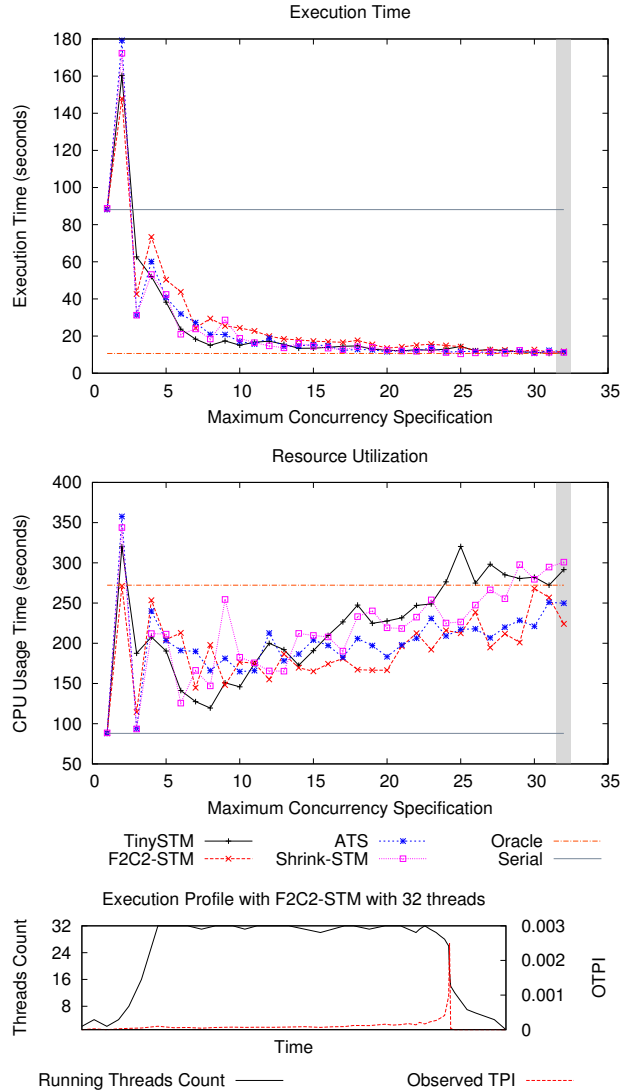


Figure 12: Execution Time, Resource Usage and Profile for *labyrinth* Benchmark well. However, F2C2-STM does exhibit slightly higher execution times when compared to ATS (average 2.5% at 32 cores) and slightly better execution time when compared to Shrink STM (on an average about 1% at 32 cores). Resource utilization on the other hand is consistently much better in F2C2-STM when compared to both ATS and Shrink STM (on an average over 12% at 32 cores). Overall, we see that concurrency control can be used on *fully scalable* workloads with acceptable overheads (even without a static specification).

The *bayes* benchmark was not used in our evaluation since it exhibited extreme

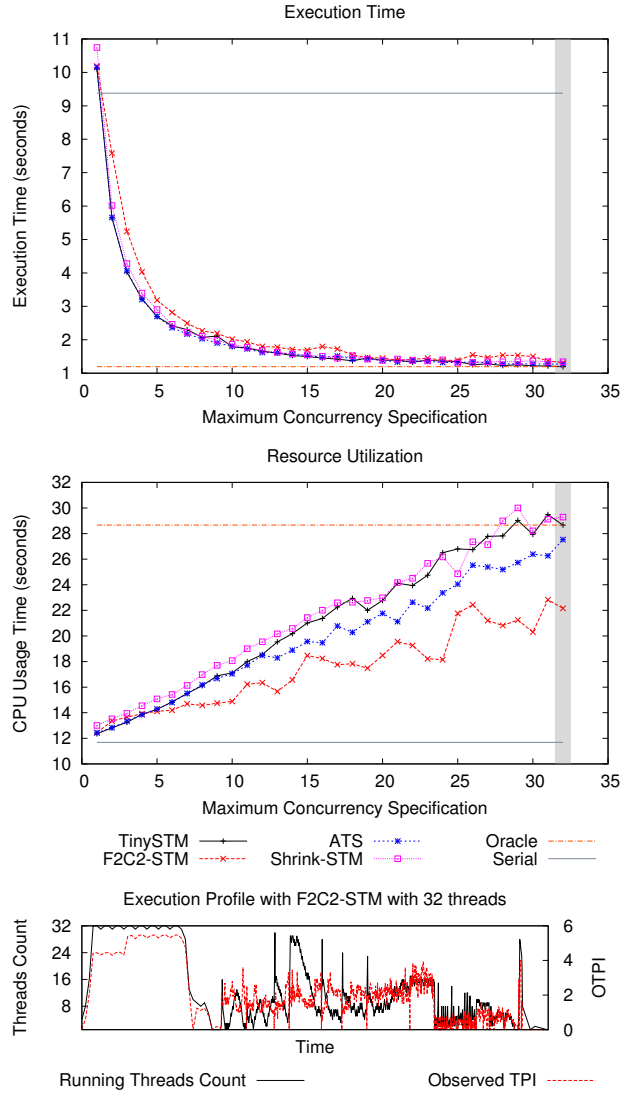


Figure 13: Execution Time, Resource Usage and Profile for *genome* Benchmark

execution time variability (baseline implementation as well) between individual runs, which made it difficult to draw conclusions (similar to observations by other researchers [31]).

2.4 Related Work

Conflict management in STMs deals with conflicts after they have occurred. Many STMs provide standard contention or conflict management techniques including priority based, age based and delay based. Many schemes in addition to these simpler

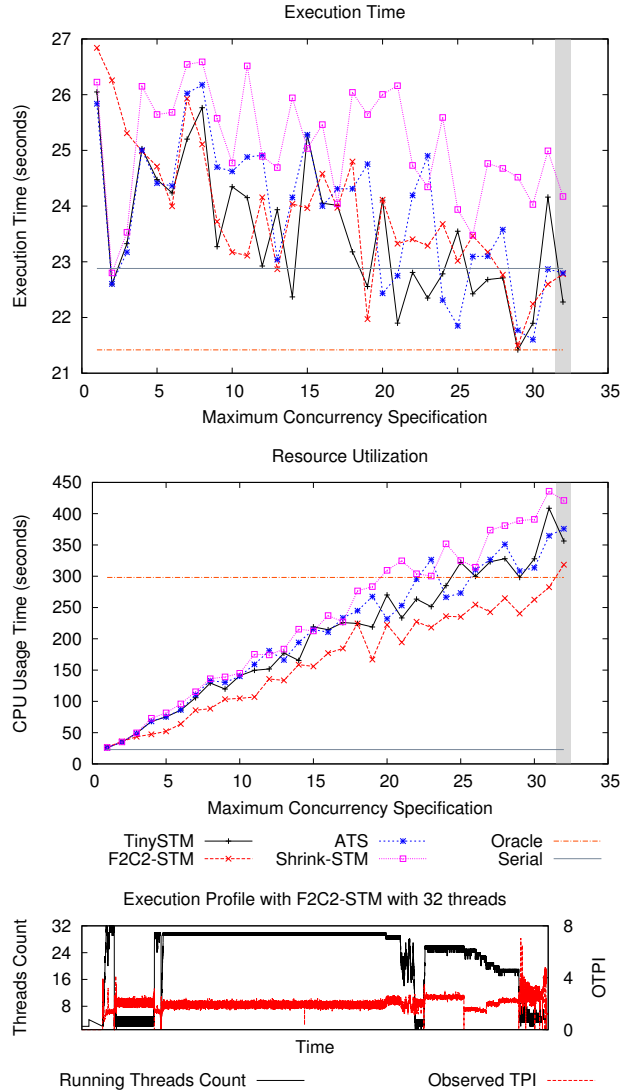


Figure 14: Execution Time, Resource Usage and Profile for *scca2* Benchmark

techniques have also been proposed [115, 40, 84, 7, 110]. Spear et al. present a contention management strategy for STMs with invisible reads [115]. [40, 84, 7] present techniques which resolve conflicts by aborting one transaction and moving it to the transaction queue of the other or similar serialization techniques. Scherer et al. introduce several contention managers including eruption, polka and kindergarten [110]. Comparative studies of contention managers have shown that different strategies work for different benchmarks and there is no clear *best* contention manager [110, 84, 63]. However, these works are orthogonal to the problem of concurrency control [84, 8]

and would benefit from our approach as well since concurrency control aims to reduce the likelihood of excessive conflicts occurring in the first place.

There have been few previous efforts which have aimed at providing concurrency control for STM applications. To the best of our knowledge, our technique outperforms them on a wider variety of applications and is significantly easier to use, requiring no code changes to applications and requiring no offline phases. Ansari et al. [6, 8] introduced a concurrency control technique which improves STAMP benchmark performance. However, their technique is not transparent to the programmer and requires re-architecting and re-implementing application code to use a thread-pool model limiting its practicality, which they mentioned as a limitation [6]. Didona et al. [37] also discuss a similar technique which requires re-architecting and re-implementing application code. Recently [104, 107] have used machine learning and modeling techniques to control concurrency in an application. However, approaches based on offline learning and modeling can degrade significantly with scenarios that challenge the set of assumptions/training data that they rely on and impose an additional burden on the programmers to train the system with appropriate datasets. Further, they are unable to completely recover the performance in the case of *scalability limited* applications nor are they able to show improvements due to dynamic adaptation. Leung et al. [80] discuss a scheme for view oriented transactional memory. However, their scheme performs concurrency control only at coarser granularities (powers of 2) and also performs global operations on each and every transaction invocation making it unsuitable for real world applications. Any technique which involves such frequent global synchronization will be unable to provide high performance, due to extremely high overheads making it unsuitable for high performance concurrency control. Chan et al. [24] also adopt global operations and hence are unable to exhibit high performance and do not make an attempt to improve resource utilization.

Yoo et al. [131] (ATS) and Dragojevic et al. [41] (Shrink STM) present state of

the art concurrency control and scheduling techniques which require no modifications to user code. ATS computes a *contention intensity* for each thread and uses this to serialize threads to control concurrency. Shrink STM introduces a novel scheduler which leverages predictions of transactional read/write sets. It identifies possible conflicts with currently running transactions using these predictions and serializes these transactions based on current contention. However, ATS typically requires an offline sensitivity phase to determine appropriate co-efficient values which makes it difficult to use in a general setting. Also, using empirically set thresholds for contention levels does not lend itself to new applications easily. Nonetheless, we also empirically demonstrate that F2C2-STM outperforms these two approaches. In addition, our work also focuses on reducing resource consumption along with improving performance and performs better in this aspect as well.

Concurrency control in the general realm of parallel processing has been shown to be beneficial. Suleman et al. [118] present concurrency control techniques for applications which are limited by off-chip bus bandwidth and data synchronization. Cheng et al. [26] propose concurrency control for applications which are hitting the memory wall. Pusukuri et al. [98] present an concurrency selection technique for applications limited by data synchronization, however they adopt an offline technique which cannot vary the concurrency level of running applications and cannot adapt to dynamic changes within an application. Jung et al. [68, 79] present techniques which balance between the number of threads and parallel execution overhead. In contrast the STM applications we focus on are not limited by memory bandwidth or data synchronization, but rather the degree of optimism (or speculation) that they exhibit. STM applications also provide stronger signals for measuring progress such as transaction throughput when compared to metrics used for traditional parallel applications such as IPC (instructions per second) which are unsuitable for our purposes (due to the notion of aborts).

2.5 Summary

While the transactional construct has significantly helped in the writing of speculative code, tuning it for best performance is still not an easy task. In this chapter we recognized that sometimes only a certain amount of speculation can be supported and discussed the need to select and adapt the degree of speculation in STM applications based on this. We presented our technique, F2C2-STM, which uses self-induced concurrency fluctuations to dynamically monitor and match varying concurrency levels in applications while minimizing global synchronization. We presented a comprehensive evaluation of F2C2-STM's performance benefits, compared it against other techniques and even demonstrate situations where it can out perform an oracle specification. We also present how F2C2-STM can effectively relieve programmers from having to indicate any concurrency specification at all. This helps the programmer in not having to think about the number of threads to use but to let the system automatically decide on the configuration with the highest performance. We believe F2C2-STM increases the utility of speculative executions using STMs providing significant performance benefits as well as simplifying the development process.

CHAPTER III

MERGE SEMANTICS FOR SPECULATION

3.1 Introduction

While, controlling the degree of speculation can provide sizable performance gains with no programmer effort, alternate ways of exposing the semantic meaning of the speculation and allowing the programmer to mitigate the impact of incorrect speculations can help scale speculative algorithms.

HPC applications, such as dense matrix applications, have seen great success in harnessing increasing amounts of parallelism. These applications have tremendous amounts of parallelism which can easily be exploited and are typically known as *regular* parallel applications.

Irregular parallel applications on the other hand are not so easy to parallelize. These applications typically rely heavily on pointer-based structures such as graphs and trees. An important characteristic of these applications is that the exact elements and therefore memory locations accessed are heavily data-dependent and can not be known until run-time. This cripples potential static analyses which are typically used to parallelize regular parallel applications. These irregular parallel applications, however, also benefit greatly from parallelization [74].

STM systems [112] can be used to speculatively parallelize irregular applications in limited circumstances. STMs provide an *atomic* construct that provides an illusion of atomicity to code executed within its scope. The programmer encompasses critical sections (called transactions) within these constructs. The STM speculatively (optimistically) executes transactions and monitors for conflicts. On detecting a conflict between a pair of transactions it will abort one of them and retry, while the other

continues. We will refer to the transaction that will abort as the *aborting transaction* and the one that will continue as the *continuing transaction*.

Software transactions provide the programmer with atomicity and isolation properties to achieve serial consistency while writing parallel code. In theory, transactions can be used to execute highly speculative algorithms. However, overheads are so high that transactions are often not used for speculation at an algorithmic level.

Overheads of STMs Two dominant sources of overheads are:

- **Overhead due to logging** All accesses to memory need to be monitored to detect conflicts.

- **Overhead due to rollbacks after a conflict**
 - **The inherent cost of rollback** There is an inherent cost of rollback that is incurred, typically due to having to restore the memory state.
 - **The cost of the lost work** The work that was executed in the aborting transaction until the point of conflict needs to be thrown away and this leads to decreased parallel efficiency.

In this chapter we target the inefficiencies due to *The cost of the lost work*. One of the key insights of this chapter is that in some applications work does not need to be thrown away when two transactions conflict but can rather be merged. In this chapter we propose a merge construct to allow programmers to salvage partially completed work in an aborting transaction, merging the states of two conflicting transactions. This has the potential to dramatically reduce overheads due to rollbacks in STM systems thus enabling programmers to write and develop highly speculative parallel algorithms.

To build on the intuition behind algorithmic speculative parallelization and the merge construct let us consider the Connected Components Problem.

3.1.1 Connected Components Problem

The Connected Components Problem is to find all the connected components in an undirected graph. Each component in the graph is to be marked with a unique component number. After detecting each individual component the number of nodes and the nodes from that component should be printed. Figure 15 shows the pseudocode for this problem.

```
// Called by the main function.
void connected_components() {
    for (int i = 0; i < nodes.size(); ++i) {
        generate_component(i)
    }
}

// Use a DFS strategy to mark all the connected nodes.
void generate_component(int i) {
    stack<int> nodes_stack;
    set<int> marked_nodes;
    nodes_stack.push(i);
    while (!nodes_stack.empty()) {
        Node* node = nodes[nodes_stack.pop()];
        // Check if the node has already been marked.
        if (node->component != -1) {
            continue;
        }
        // Mark the node with the component number.
        node->component = i;
        marked_nodes.insert(node->id);
        for (each neighbor of node) {
            nodes_stack.push(neighbor);
        }
    }
    print("Component number:", i, " size:", marked_nodes.size(), " nodes:",
        marked_nodes);
}
```

Figure 15: Pseudocode for connected components problem

The code is fairly straight forward and employs a DFS strategy to detect each component. This algorithm cannot be parallelized in the traditional data parallel manner. Each iteration of the loop is dependent on the previous iteration and the execution is highly data dependent. This cripples any static parallelization techniques. There are specialized parallel DFS algorithms [44, 3, 4]. However, these parallel algorithms are typically complicated and not in the area of expertise of the programmer.

3.1.2 Speculatively Parallelizing Graph Algorithms

There is an extremely simple and intuitive method to speculatively parallelize some graph algorithms using STMs. The *basic idea is to speculatively launch multiple parallel threads in different parts of the graph, each executing the same code as the sequential version* as software transactions. These threads run normally until they conflict with another thread. On a conflict the STM system kicks in and aborts one of the transactions, thus leading to a correct parallelization.

To elaborate, consider graph applications which traverse the edges of a graph performing some arbitrary operation on the nodes (the connected components problem, is one such application). We can speculatively launch multiple threads at different parts of the graph (see Figure 16) each executing the same code as the sequential version. If it turns out that the threads were speculatively launched on disconnected parts of the graph (see Figure 16(a)), they will not conflict and this approach leads to a correct parallelized execution. However, if the speculatively launched threads operate on the same component they will eventually conflict (and our STM system performs a rollback on one of them).

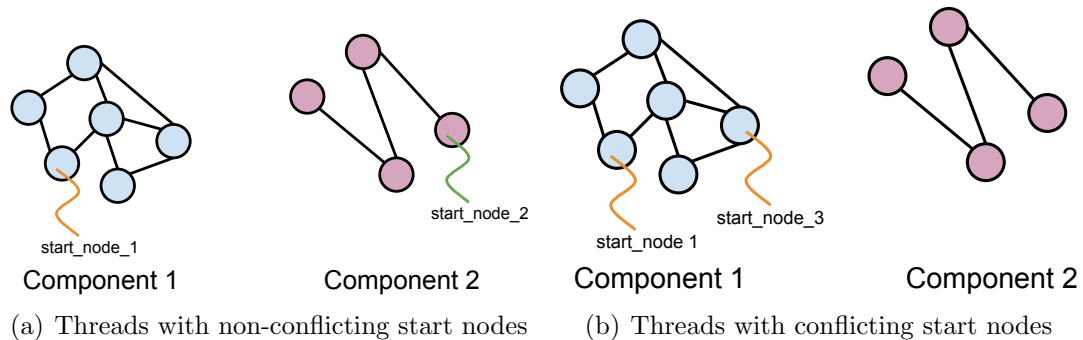


Figure 16: Threads with different start points

This is a very powerful notion, since we have obtained a sort of data parallelism over an irregular data structure. This type of data parallelism is much easier for the programmer to reason about and deal with when compared to specialized parallel algorithms. As long as a sufficient number of speculative threads operate on disjoint

parts of the graph we can attain a speedup. On the flip side, each time two threads conflict, the STM aborts one of them and its work is lost leading to a less efficient solution.

This type of speculative parallelization is susceptible to the typical scalability concerns of STM applications when there are a large number of conflicts. A measure of scalability is parallel efficiency, P_e , defined as:

$$P_e = S / (p * T(p))$$

Where S represents the wall clock time of a sequential execution, p is the number of processors and $T(p)$ is the wall clock time of an execution on p processors. As the number of conflicts increase, there is an increase in overhead due to both the loss of useful work and time spent in servicing these conflicts, consequently increasing $T(p)$.

Instead of discarding the work performed by the aborting transaction if we can *merge* it with the continuing transaction, we would get a significant reduction in the overhead due to conflicts, thereby decreasing $T(p)$ and consequently increasing P_e . This enables the execution of highly speculative algorithms efficiently without performance loss due to mis-speculation. We believe the *merge* construct is key in enabling speculative parallelization to achieve performance gains. Algorithms which have potential for speculative parallelization such as connected components are often used as kernels in a wide variety of areas such as video processing [66], image retrieval [32], traffic monitoring [15], object recognition in 3D images [23] and many more. The merge construct can not be applied to all STM based applications. We discuss the properties of the merge construct in Section 3.4.

The remainder of this chapter is organized as follows, in Section 3.2 we discuss the merge construct in detail. In Section 3.3 we discuss the API. We discuss the properties of the merge construct in Section 3.4. In Section 3.5 we discuss the benchmarks, experimental results and observations. We discuss related work in Section 3.6, then conclude and present future work.

3.2 Merge Construct

In this section we'll discuss the merge construct in detail using the connected components problem introduced in the previous section. Figure 15 shows the pseudocode for this problem. This problem is also amenable to the type of speculative parallelization discussed in the previous section, i.e. speculatively launching threads on different parts of the graph (as transactions) and relying on an STM to rollback on conflicts. If at least a few of the speculatively launched threads start in disjoint components we would have obtained some amount of parallelism. The underlying reason why no parallelism is obtained when two transactions conflict is because the work performed by the aborting transaction is thrown away. We will now discuss how to merge the work performed by the aborting transaction into the continuing transaction using the merge construct.

The merge construct consists of user defined MERGE and UPDATE functions and a set of APIs exposed by our framework: MAKE_AVAILABLE, SAFE_UPDATE_POINT and SNAPSHOT (optional).

MERGE The job of the MERGE function is to take the work from the aborting transaction and put it into the continuing transaction. A preliminary MERGE function for the connected components problem is shown in Figure 17 (we will refine this as we go along). The essence of the MERGE function is extremely simple. It simply takes the *nodes_stack* and the *marked_nodes* from the aborting transaction and adds it to the corresponding data structures in the continuing transaction.

```
merge(transaction t1, transaction t2) {  
    t1.nodes_stack.add(t2.nodes_stack);  
    t1.marked_nodes.add(t2.marked_nodes);  
}
```

Figure 17: Preliminary pseudocode for the MERGE function.
t1 = continuing transaction, *t2* = aborting transaction.

Our framework automatically invokes the user defined MERGE function once on

every conflict. The state of the aborting transaction is guaranteed to still be in tact during the execution of MERGE. MERGE is executed by the aborting thread before it terminates. Figure 18 shows the execution schedule of two conflicting transactions: $t1$ and $t2$.

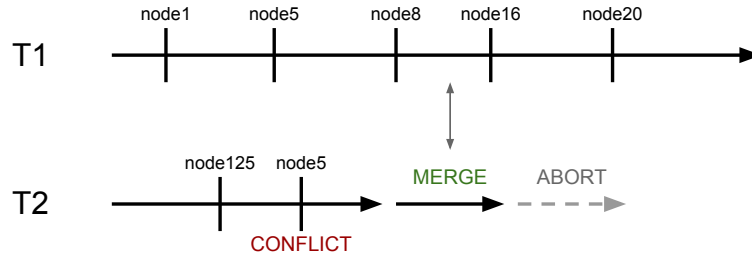


Figure 18: Transaction Schedule for T1 and T2

As you can imagine, all kinds of weird race conditions can arise with such a simple implementation. We have to deal with two main issues. Firstly, we need to ensure that the *nodes_stack* and *marked_nodes* in the aborting transaction ($t2$) are consistent when MERGE is executed (subsection: 1. Consistency). Secondly, MERGE needs to be able to access the *nodes_stack* and *marked_nodes* in the continuing transaction ($t1$) in a synchronized manner (subsection: 2. Synchronization).

1. Consistency To tackle the first issue let us define the state, DS , of the aborting transaction ($t2$) at any point in its execution. We define DS as the union of all the data structures from the aborting transaction used in the MERGE function. In our example:

$$DS = (\textit{nodes_stack}, \textit{marked_nodes}).$$

Note, that we are only concerned about the semantic state of these data structures and not their actual in-memory representation. For example, let's say the *marked_nodes* set contains integers 1 and 2. Whether it is stored as (1, 2) or (2, 1) internally is not relevant to us and they are semantically equivalent. For our purposes, DS completely represents the state of the aborting transaction with respect to the merge construct. DS can either be a:

Valid State is defined as a state, in which all the data structures in *DS* are currently consistent with each other and are safe to use in the MERGE function.

Invalid State is defined as a state in which the data structures in *DS* are currently inconsistent with each other and are not safe to use in the MERGE function.

When MERGE is executed, if *t2*'s *DS* is in a valid state a consistent set of data structures are available. However, if *DS* is in an invalid state an inconsistent set of data structures will be available and can not be used by MERGE.

To get a better understanding of what valid states are let us first see how an invalid (inconsistent) state might arise in MERGE. Consider again the pseudocode from Figure 15. The only lines in the pseudocode which can potentially cause an STM conflict (and hence cause MERGE to be executed) are those with read/write operations on global memory. In particular, lines 16 and 20 read the component number of a node from global memory and write to it. Only at these two lines can a conflict potentially occur. Figure 19(a) shows this snippet again for convenience.

```

// Check if the node has already been marked.
if (node->component != -1) {
    continue;
}
// Mark the node with the component number.
node->component = i;
marked_nodes.insert(node->id);
for (each neighbor of node) {
    nodes_stack.push(neighbor);
}

```

(a) Original code

```

// Check if the node has already been marked.
if (node->component != -1) {
    continue;
}
// Mark the node with the component number.
marked_nodes.insert(node->id);
node->component = i;
for (each neighbor of node) {
    nodes_stack.push(neighbor);
}

```

(b) Modified code

Figure 19: Code snippets

Figure 19(b) shows the same code, but with lines 20 and 21 interchanged in position. While this alternate implementation is still a correct sequential implementation

and a correct parallel STM implementation it can lead to an invalid state while executing MERGE. Let's see how. Potential STM conflicts can now occur on either line 16 or line 21 (Figure 19(b)). Let's say a conflict occurs on line 21 while processing $node_a$. $node_a$ would have already been added to the *marked_nodes* set, semantically indicating that the node has already been processed. However, since the conflict occurs on line 21 and its execution ceases at that point, $node_a$'s neighbors will not be inserted into *nodes_stack*. Once MERGE (Figure 17) executes after the conflict on line 21 it will append the contents of $t2$'s data structures into $t1$'s. $t2$'s *marked_nodes* indicates that $node_a$ has been completely processed, while its *nodes_stack* indicates that $node_a$ has not been processed. Now, when the continuing transaction $t1$ continues executing its code, there is a possibility that all the nodes which are neighbors of $node_a$ get skipped since they were never added to the traversal stack *nodes_stack* in the aborting transaction. This is an incorrect execution due to an invalid state.

While this example is contrived and it is simple to see that in the natural expression of the transaction the states are in fact always valid during MERGE, it might be the case that for some transactional applications it is extremely difficult to write code such that state (DS) is valid during MERGE execution. For such situations we provide the programmer with a SNAPSHOT API (described shortly).

To determine if a given piece of code will result in a valid state or invalid state during the execution of MERGE the programmer needs to:

a. Identify conflict points Conflict points are those lines in the code which read/write to global memory. Depending on the STM system that is being used these statements are typically wrapped in special STM API calls. Hence, this is normally an easy step.

b. Identify the possibility of an Invalid State The programmer needs to take each of the conflict points identified in the previous step and check if executing the MERGE function after that conflict point will lead to an invalid state in MERGE. In

our example, this is straightforward, since all changes to structures in DS are made after all potential conflict points. No invalid states can arise. However, it might not be so simple to make this determination for all applications. To deal with these applications we provide the programmer with the SNAPSHOT API.

A call to the SNAPSHOT API can be inserted by the programmer at any point that the data structures will be consistent in the main transaction code. This API takes a checkpoint of the current state of the data structures and ensures that only this state is accessed in any subsequent invocation of MERGE, thereby alleviating any concerns of invalid states. It is typically extremely simple to determine where to make the SNAPSHOT call, normally at the beginning or end of the loop or function call. For example, in our example it could be placed as the first line in the loop (see line 8 in Figure 20). Note that the SNAPSHOT call is not required in our implementation since valid states are guaranteed but is shown here for completeness.

Now that we have guaranteed that the aborting transaction's DS is in a valid state, we shift our focus to the issue of synchronization with the continuing transaction in MERGE.

2. Synchronization Consider again the execution schedules of two conflicting transactions, $t1$ and $t2$ in Figure 18.

In Figure 18, once $t2$ encounters a conflict, it will execute MERGE. MERGE needs access to the private states of both $t1$ and $t2$. In the previous section we discussed how we can guarantee that $t2$'s data structures (DS) are in a valid state. However, while MERGE is executing, $t1$ will be executing simultaneously and its data structures will be undergoing modifications. To overcome this, we allow the programmer to provide merge specific data structures for $t1$ in the transaction and identify them using our API. Only these merge specific data structures should be used in the MERGE code. Our framework guarantees that these merge specific data structures will only be used

in a synchronized manner (i.e. with appropriate locking).

In our example MERGE code (Figure 17) instead of directly inserting into $t1$'s *nodes_stack* and *marked_nodes* set, merge specific data structures should be used. See lines 34 - 36 in Figure 20 for the final MERGE code.

UPDATE Since MERGE inserts into merge specific data structures, the continuing transaction $t1$ needs to incorporate this information into its actual data structures. To enable this, the programmer provides the second user-defined function: UPDATE. This function simply takes the information from the merged data structures and incorporates it into the actual data structures (lines 26 - 32 in Figure 20).

Our framework provides a SAFE_UPDATE_POINT API which can be called periodically by the transaction. Our framework keeps track of when MERGEs are performed and uses these SAFE_UPDATE_POINT calls as opportunities to invoke the UPDATE function. The UPDATE function is executed by $t1$ itself.

Let's recap by looking at the complete, final code sample in Figure 20.

Note that in the UPDATE function, by virtue of marking all the merged nodes, locks are acquired by the STM system on all of the newly added nodes. The MAKE_AVAILABLE API call allows the programmer access to any local variables inside the MERGE and UPDATE functions.

In the context of the connect components problem, while there is a significant performance improvement due to this kind of speculative parallelization technique using STMs, there is no performance improvement attained due to the merge construct. This is because the DFS algorithm is as quick as the merge function itself. We use this example to motivate our ideas. While the merge does not provide a performance benefit in this application, it is still applicable to other applications such as the Minimum Spanning Tree (MST) problem. This is discussed in more detail in Section 3.5.1. Similar modifications are made in the MST Problem and we demonstrate significant

```

void generate_component(int component_number) {
atomic {
    stack<int> nodes_stack, merged_nodes_stack;
    set<int> marked_nodes, merged_marked_nodes;
    MAKE_AVAILABLE(nodes_stack, marked_nodes, merged_nodes_stack, merged_marked_nodes,
        component_number);
    nodes_stack.push(i);
    while (!nodes_stack.empty()) {
        SNAPSHOT(nodes_stack, marked_nodes); // OPTIONAL
        Node* node = nodes[nodes_stack.pop()];
        // Check if the node has already been marked.
        if (node->component != -1) {
            continue;
        }
        // Mark the node with the component number.
        node->component = component_number;
        marked_nodes.insert(node->id);
        for (each neighbor of node) {
            nodes_stack.push(neighbor);
        }
        SAFE_UPDATE_POINT();
    }
    print("Component_number:", i, " size:", marked_nodes.size(), " nodes:",
        marked_nodes);
}
}

UPDATE(transaction t) {
    t.nodes_stack.add(t.merged_nodes_stack);
    t.marked_nodes.add(t.merged_marked_nodes);
    for (node in t.merged_marked_nodes) {
        node.component = t.component_number;
    }
}

MERGE(transaction t1, transaction t2) {
    t1.merged_nodes_stack.add(t2.nodes_stack);
    t1.merged_marked_nodes.add(t2.marked_nodes);
}

```

Figure 20: Complete pseudocode for connected components with *merge*

speedups due to *both* the speculative STM parallelization *and* the merge construct as well as sustained scalability at higher core counts. The key behind the improvement being the dramatic reduction in redundant work that needs to be performed due to the merge construct.

3.2.1 STM Requirements

Our framework requires the STM system to satisfy certain requirements. Some modifications need to be made to the underlying STM as well. Here are the requirements of the STM system:

- 1. Detect conflicts early** The STM system must be capable of detecting conflicts

and rolling back as soon as a conflict is detected. In a write through system locks are typically acquired during the write itself and hence conflicts will be detected early. In a write back system there are two popular locking schemes:

a. Commit Time Locking (CTL) postpones acquisition of the locks till commit time. Our model cannot use this kind of a locking scheme since conflicts need to be detected early, not during commit time. Detecting conflicts at commit time will negate any potential benefit of preventing the execution of redundant work.

b. Encounter Time Locking (ETL) In write back encounter time locking, though the writes to memory happen only during commit time, locks are acquired during the execution of the transaction itself, hence allowing conflicts to be detected early.

2. No-retry transactions The STM system must be capable of not retrying aborted transactions.

Most importantly we need either a write through or a write back ETL STM system to use our framework. We built our framework on top of TinySTM [121] and found it was straight-forward to make any additional changes. Another minor change we had to make is that TinySTM yields to the conflicting transaction before retrying, as a performance improvement technique. Since we do not retry after aborts we removed the option to yield.

3.3 Framework API

In this section we propose an API to support the merge construct and briefly discuss the implementation of our prototype. The five components of the API are summarized in Table 1.

MAKE_AVAILABLE This API can be called with an arbitrary number of arguments and each of the arguments becomes available for the programmer to use in the

Table 1: Summary of API

API	Required	Summary
MAKE_AVAILABLE	Yes	Makes arguments available in MERGE and UPDATE functions.
MERGE	Yes	User defined function called by aborting transaction to merge information.
SAFE_UPDATE_POINT	Yes	Indicates point(s) in code where it's safe to invoke UPDATE.
UPDATE	Yes	User defined function called by continuing transaction to updates data structures.
SNAPSHOT	No	Creates copy of arguments for use in MERGE to ensure a valid state.

MERGE and UPDATE functions. For example, calling:

```
MAKE_AVAILABLE(value1, value2);
```

will make *value1* and *value2* accessible inside the functions as *t1.value1* and *t1.value2*.

MERGE function is the user defined function which merges information from the aborting transaction. It provides to the programmer the two conflicting transactions *t1* and *t2* as input.

```
MERGE(transaction t1, transaction t2);
```

t1 is the continuing transaction and *t2* is the aborting transaction. Variables *t1* and *t2* provide access to the private members made accessible through the MAKE_AVAILABLE call (example: *t1.value*). This function is executed by the aborting thread.

SAFE_UPDATE_POINT can be called by the programmer to specify safe update point(s) in the transaction (single or multiple times). This is a simple function call:

```
SAFE_UPDATE_POINT();
```

These point(s) mark where in the transaction it is safe to incorporate the merge specific data structures into the main data structures. Our framework will call the UPDATE function only when a merge has actually been performed irrespective of the number of times this API gets called during execution.

UPDATE function is the user defined function that allows the programmer to incorporate the results from the merge specific data structure into the main data structures in a safe, synchronized manner.

```
UPDATE(transaction t);
```

Variables made available through `MAKE_AVAILABLE` are accessible inside the function as members of t (example: $t.value$). This function is executed by the continuing thread.

SNAPSHOT This is an optional API. As discussed previously (Section 3.2) it might be difficult for some applications to ensure that the aborting transaction is in a valid state during execution of the `MERGE` function. In such cases the programmer can simply insert a call to the `SNAPSHOT` API. Our framework will create a snapshot of the data structures specified as arguments. In all subsequent `MERGE` invocations, the snapshot created by the latest `SNAPSHOT` call will be used. This API can take an arbitrary number of arguments as input. For example:

```
SNAPSHOT(value1, value2);
```

This will create copies of $value1$ and $value2$. All calls to `MERGE` will use the latest snapshot values for $value1$ and $value2$ instead of their current values. Creating `SNAPSHOT`s will indeed increase the execution time since copies need to be made. However, the overall improvements we gain due to merging far outweigh this overhead (see Section 3.5).

Mutual Exclusion The `MERGE` and `UPDATE` functions are synchronized. Our framework uses a set of per transaction locks to ensure mutual exclusion and guarantee correctness. When `MERGE` is called our framework's internal locks on both transactions are automatically acquired (order is based on the transaction id which prevents deadlocks). When `UPDATE` is called our framework's internal lock on the continuing transaction is automatically acquired. This ensures that accesses to shared data members are safe.

A transaction is not allowed to abort when another transaction is `MERGE`-ing with it. This is ensured by acquiring the internal lock on the transaction before aborting. Further, if two transactions encounter conflicts at the same time, the order

of lock acquisition determines order of MERGE-ing. Our prototype assumes only one type of transaction in the code. To deal with multiple non-composable transactions we can name each distinct transaction with a simple type and use that to type the UPDATE and MERGE functions.

3.4 *Properties*

In this section we discuss the properties of the merge construct considering applications which have been speculatively parallelized using the technique discussed in Section 3.1.2. For the merge construct to operate as expected it must be *Correct* and *Efficient*.

Correctness The merge construct should be correct. That is to say that after merge is performed the application must be capable of continuing to execute normally. The result obtained after performing a merge should be the same as that, that would have been obtained without a merge. Formally, let the transaction T be operating on the graph G and its result represented by $T(G)$. Consider two instances of T : T_1 and T_2 operating on two disjoint sections of the graph G_1 and G_2 . G_1 and G_2 evolve with time, as the transactions execute. If on continued execution of T_1 and T_2 the disjoint sections G_1 and G_2 intersect the MERGE function M is invoked. M is invoked with the arguments (T_1, T_2) where T_2 is the aborting transaction and T_1 is the continuing transaction. Let the operation defined by M be \odot . To be correct the following property must be satisfied:

$$T(G_1 \cup G_2) = T_1(G_1) \odot T_2(G_2)$$

Where $=$ is a semantic equivalence and $G_1 \cup G_2$ is simple union of the graphs. This guarantees that the execution result after the merge is the same as that if there had been no merge.

Efficiency In addition to being *correct* merge must also be *efficient*. To obtain a performance improvement over an execution without merge, \odot must be more efficient than running T itself. If T_2 is being aborted and it has operated on G_2 so far then:

$$t_e(T_1(G_1) \odot T_2(G_2)) < t_e(T(G_2))$$

Where $t_e(x)$ represents the execution time of x . In other words, it must be quicker to re-use the discarded work than to re-execute it in transaction T_1 .

3.5 *Experimental Evaluation*

In this section we demonstrate the benefits of our approach through the Connected Components benchmark as well as the Minimum Spanning Tree benchmark.

All experiments were performed on a dual quad-core Intel Xeon E5540 (2.53GHz) machine running Ubuntu 10.10 using up to 8 concurrent threads. The benchmarks were compiled using GCC 4.4.5 (Ubuntu/Linaro 4.4.4-14ubuntu5) with the O3 flag set. OpenMP was used to parallelize the code. We used TinySTM 1.0.0 to protect the transactions. All results were obtained by averaging the results of 5 executions. TinySTM was configured with write back ETL (Section 3.2.1). Our results compare:

- Serial execution without any parallel overheads (No STM overhead)
- Parallel execution using STMs
- Parallel execution using STMs and *merge*

Datasets The input datasets for both the benchmarks were randomly generated undirected graphs. We parametrize the graph generation process using T , N , and X . Each graph is a forest containing T base trees. Each node in a tree has a random number of neighbors, selected uniformly from $[0, N)$. If 0 is selected, it has no neighbors and the number of trees in the graph hence increases by 1. There are a total of X nodes in the graph which are evenly divided between the number of base trees.

3.5.1 Minimum Spanning Tree

Given a weighted undirected graph, the minimum spanning tree algorithm computes a tree from the graph whose weight is less than or equal to the weight of every other spanning tree for the graph. More generally, any undirected graph has a minimum spanning forest which is a union of the disjoint minimum spanning trees.

We use the same technique as described in Section 3.1.2 to parallelize the Minimum Spanning Tree benchmark. First with STMs to achieve parallelization over the serial execution and then adding the merge construct to provide even better performance. In this benchmark transaction durations are much larger than that of the connected components benchmark and the merge construct almost always provides consistent performance improvement.

The benchmark is implemented using Prim's algorithm. The benchmark does not use the fastest MST algorithm available but uses a simple algorithm that a novice programmer might use. Figure 21 gives the pseudocode including the complete merge construct. The *get_next_edge* function does a simple linear scan over the *marked_nodes* set and looks for an adjacent node which has not yet been marked. We also demonstrate the performance impacts of the SNAPSHOT feature for this benchmark. The natural expression of the code as shown in Figure 21 does not need SNAPSHOT to behave correctly but we use it to study its effects on performance.

The datasets used for the Minimum Spanning Tree benchmark are (methodology was described at the beginning of Section 3.5).

- DS1 (6000 nodes): $X = 6000, T = 6, N = 6$.
- DS2 (9000 nodes): $X = 9000, T = 5, N = 6$.
- DS3 (12000 nodes): $X = 12000, T = 4, N = 8$.
- DS4 (16000 nodes): $X = 16000, T = 3, N = 8$.

```

void prims(int i) {
atomic{
  int tree_number = i;
  set<Pair*> tree_edges , merged_tree_edges;
  set<int> marked_nodes , merged_marked_nodes;
  MAKE_AVAILABLE(tree_edges , marked_nodes , merged_tree_edges , merged_marked_nodes ,
    tree_number);
  int next_node = i;
  while (true) {
    SNAPSHOT(tree_edges , marked_nodes); // OPTIONAL
    Node* node = nodes[next_node];
    if (node->component != -1) {
      return;
    }
    node->tree_number = tree_number;
    marked_nodes.insert(node->id);
    Pair* edge = get_next_edge(&marked_nodes);
    if (edge == NULL) {
      break;
    }
    next_node = edge->node2->id;
    tree_edges.insert(edge);
    SAFE_UPDATE_POINT();
  }
  print("Tree_number:", tree_number , " size:" , marked_nodes.size() , " nodes:" ,
    marked_nodes);
}
}

UPDATE(transaction t) {
  t.tree_edges.add(t.merged_tree_edges);
  t.marked_nodes.add(t.merged_marked_nodes);
  for (node in t.merged_marked_nodes) {
    node.tree_number = t.tree_number;
  }
}

MERGE(transaction t1, transaction t2) {
  t1.merged_tree_edges.add(t2.tree_edges);
  t1.merged_marked_nodes.add(t2.marked_nodes);
}

```

Figure 21: Pseudocode for Prim’s with merge

DS1 is a smaller dataset with lower contention. DS2 and DS3 are datasets with larger sizes and increasing contention. DS4 is the largest dataset with the most contention. The results we obtained are reported in Figure 22.

Observations and Discussion Parallelizing the application using the simple STM parallelization technique (Section 3.1.2) provides good speedups. In all the datasets: DS1, DS2, DS3 and DS4 all scenarios with more than 1 thread run faster than the serial version. In the STM implementation (without merge), as we increase the number of threads, execution time decreases as expected. However, in the case of DS3 and DS4 with 8 threads there is a slow down when compared to 4 threads.

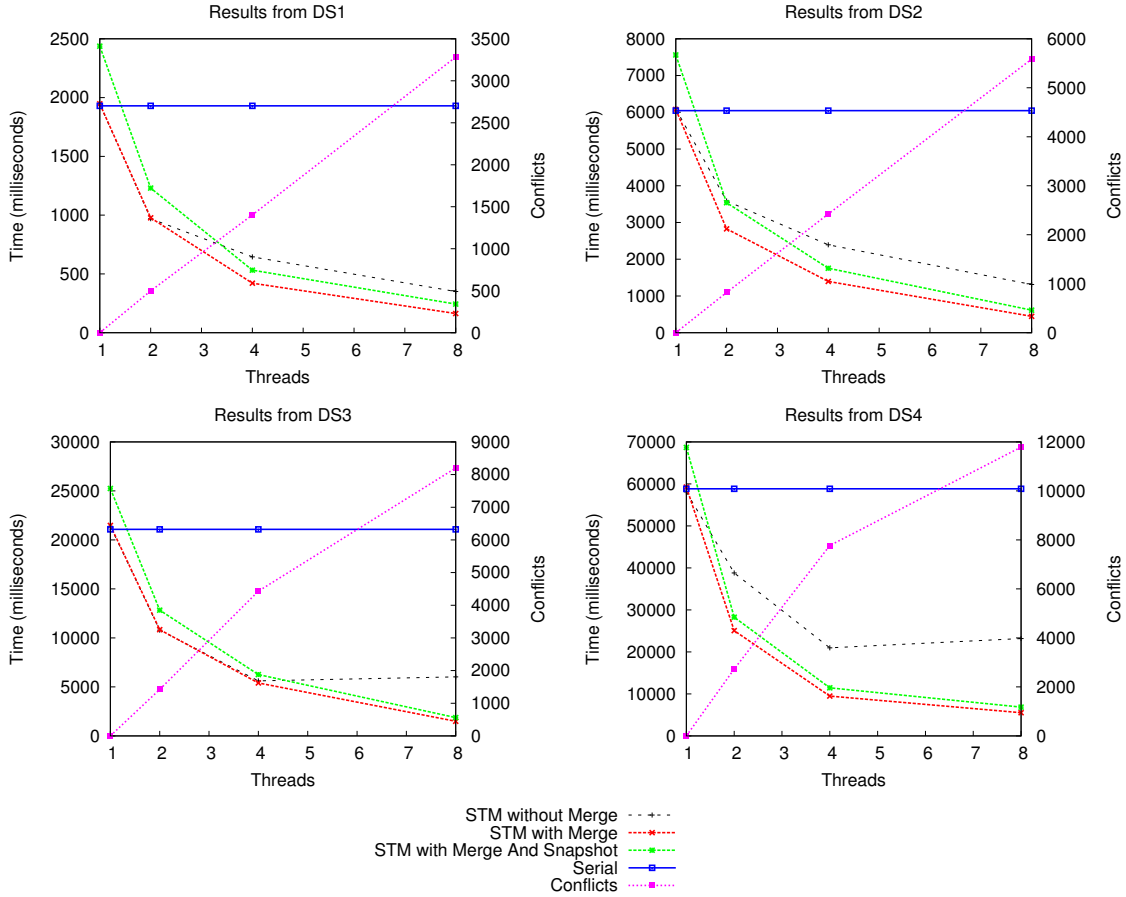


Figure 22: Results of the Minimum Spanning Tree benchmark

This is due to higher conflict overheads with an increased number of threads. In fact, this starkly depicts one of the main criticisms of STMs, that they do not scale very well with increasing contention. The merge construct is able to alleviate this issue completely and demonstrates sustained scalability with an increasing number of threads.

We report the results of two implementations of the MST benchmark using merge: one without SNAPSHOT-ing and one with SNAPSHOT-ing. Recall that for our benchmark the SNAPSHOT is an optional call. As expected the version without SNAPSHOTs performs better. This is simply due to lower overhead of not having to constantly create safe snapshots. At lower thread counts, the version without SNAPSHOT consistently performs better than the simple STM parallelization. And

very importantly it also scales excellently as the number of threads increase. This can be attributed to the fact that the conflict overhead is reduced significantly as there is minimal wasted work when a transaction is aborted, as much of it can be re-used. The version with SNAPSHOT-ing performs slightly worse than the version without SNAPSHOT-ing but at higher thread counts even this version performs significantly better than the simple STM parallelization. The speedups are significant. For DS1 at 8 threads, the STM parallelization with merge is more than 50% faster than the simple STM parallelization and more than 90% faster than the serial version. The parallel efficiency P_e is surprisingly 1.48 (we will explain the super linear speedup in the next paragraph). For DS4 at 8 threads, the STM parallelization with merge is more than 75% faster than the simple STM parallelization and more than 90% faster than the serial version. The parallel efficiency P_e is 1.33. Again, a super linear speedup when compared to the linear version. DS2 and DS3 also exhibit very similar characteristics.

The super linear speedups are due to the fact that the *get_next_edge* function does a simple scan over the *marked_nodes* set, looking for other unmarked nodes. This simple lookup results in a $O(n^2)$ scan over the *marked_nodes* set. On parallelization, the *marked_nodes* set actually gets divided between the many competing threads. Therefore the algorithm runs much more quickly due to smaller set sizes. This results in the super linear speedups that we observe. The super linear speedup does not appear in the standard STM parallelization (without merge) as this advantage of splitting the *marked_nodes* set between threads can never be materialized as there is no functionality to merge them.

3.5.2 Connected Components

As discussed in Section 3.2, the speculative parallelization approach provides performance improvements for this benchmark while the merge construct does not. This is

simply because the merge construct is as expensive as the DFS search itself. However, if in addition to a simple DFS, if the application needed to perform some arbitrary processing at each node, then the merge construct would indeed be beneficial.

We still present our results with the Connected Components benchmark to demonstrate the properties of the merge construct, especially with varying transaction durations. To better understand how the merge construct performs with transactions of varying duration (perhaps due to extra processing which needs to be performed at each node of the graph) we simulate transactions of varying duration in the benchmark and report these results in Figure 23(a) and Figure 23(b).

The pseudocode for the Connected Components benchmark is given in Figure 20.

We used two datasets DS1 and DS2 to generate the graphs. The exact parameters used to generate DS1 and DS2 are:

- DS1 (80000 nodes): $X = 80000$, $T = 4$, $N = 8$.
- DS2 (500000 nodes): $X = 500000$, $T = 2$, $N = 6$.

Observations and Discussion Figure 23(a) and 23(b) follow the same pattern. Each shows the execution time of the connected components application with an increasing amount of transaction processing time inserted into it. DS1 is a smaller dataset containing 80000 nodes (with lower number of conflicts) while DS2 is bigger at 500000 nodes (with higher amount of conflicts).

In Figure 23(a)(i) the parallel STM implementation without merge scales as expected and it runs quicker than the serial implementation when there are more than 2 threads. This demonstrates that this method of parallelization is effective, yet simple to achieve. As discussed, the merge construct does not provide any performance benefits in this case simply because the merge is as expensive the original DFS. However, if we were to consider any application which also performed some processing at each node, performance gains are obtained. Figures 23(a)(ii), 23(a)(iii) and 23(a)(iv)

show the performance gains when we introduced processing at each node (in the form of a busy loop) for 10, 20 and 30 microseconds. Even with such small transaction durations, we see sizable speedups. At 8 threads we get performance gains of 23%, 33% and 37%. The performance gains increase as we increase the duration of the transaction. The same pattern is observed in Figure 23(b).

We also observe higher performance benefits of using merge when there are more conflicts. This is expected as this causes a higher number of aborted transactions and merge removes much of the penalty of a transaction aborting. Comparing Figure 23(a) and 23(b) we see that with the increased contention of DS2 we get much better performance gains using merge.

3.6 Related work

Overhead reduction in STMs is an active area of research. [2] tries to reduce overheads through static analysis. [56] deals with conflicts at an abstract data type (ADT) level. [27] tries to decrease the number of conflicts by predicting data access patterns. We on the other hand, try to decrease the severity of conflicts. Many other techniques have been developed (see [76] for a recent list). Our work is orthogonal to the other work and they can benefit from our approach as well.

Much work has also gone into creating new parallel algorithms for problems such as the minimum spanning tree [67, 12], the connected components problem [57, 43] and DFS [44]. However our approach maintains its simplicity and generality making it available to any programmer irrespective of background.

[16] proposes a mechanism to run "twilight" code at end of a transaction before it commits/aborts. While this mechanism allows the programmer to correct errors before committing, it does not prevent the execution of redundant work and re-use of work between threads which is the main source of speedups in our framework.

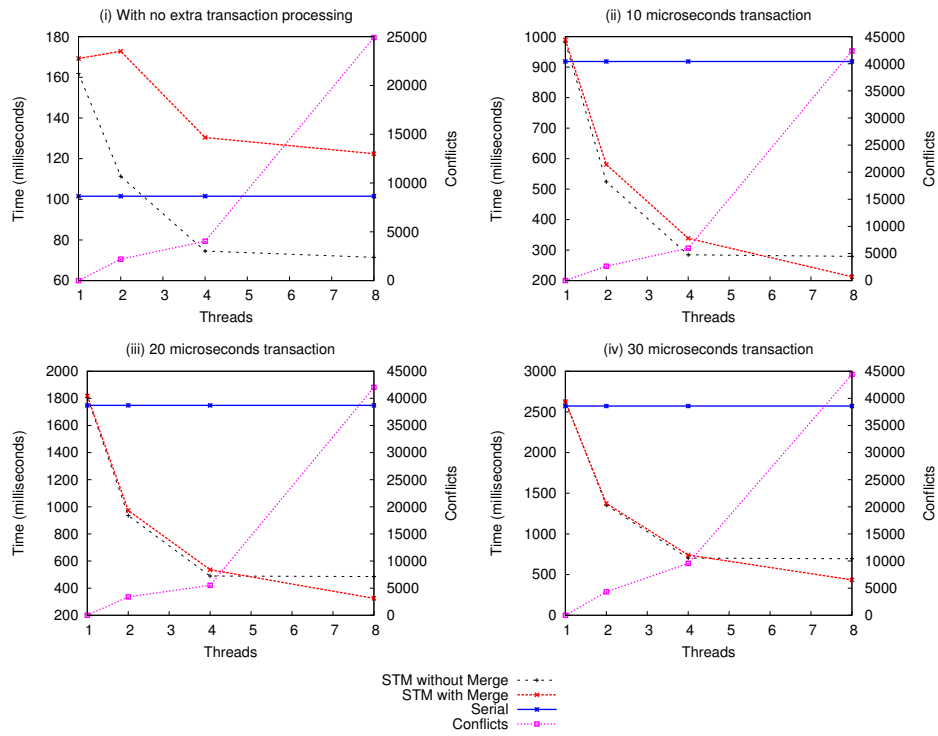
[71] discusses a parallel speculative algorithm for Minimum Spanning Tree (MST);

the main focus of the paper is on developing a parallel algorithm for MST that uses data merging to improve execution. Although the paper attempts to leverage the idea of merge, it does not concern itself with how to extend transactional memory to support the merge construct as a generalized abstraction for supporting and promoting aggressive speculation. Moreover, their implementation of the MST falls short on performance, even though it's scalable, they are unable to demonstrate speedups over a serial implementation. The main contribution of this chapter in contrast is to propose APIs which programmers can use to leverage partially completed transactions and merge the results so that partially completed work is not wasted. We also study the properties of the merge construct and present a detailed empirical evaluation. One of the biggest impediments to the use of transactional memory in highly speculative computation is the high overhead of rollbacks and restarts. Such high overheads dissuade algorithm designers from speculating aggressively. This chapter proposes the critical merge construct to avoid such costly rollbacks and restarts thus promoting the design of highly speculative algorithms.

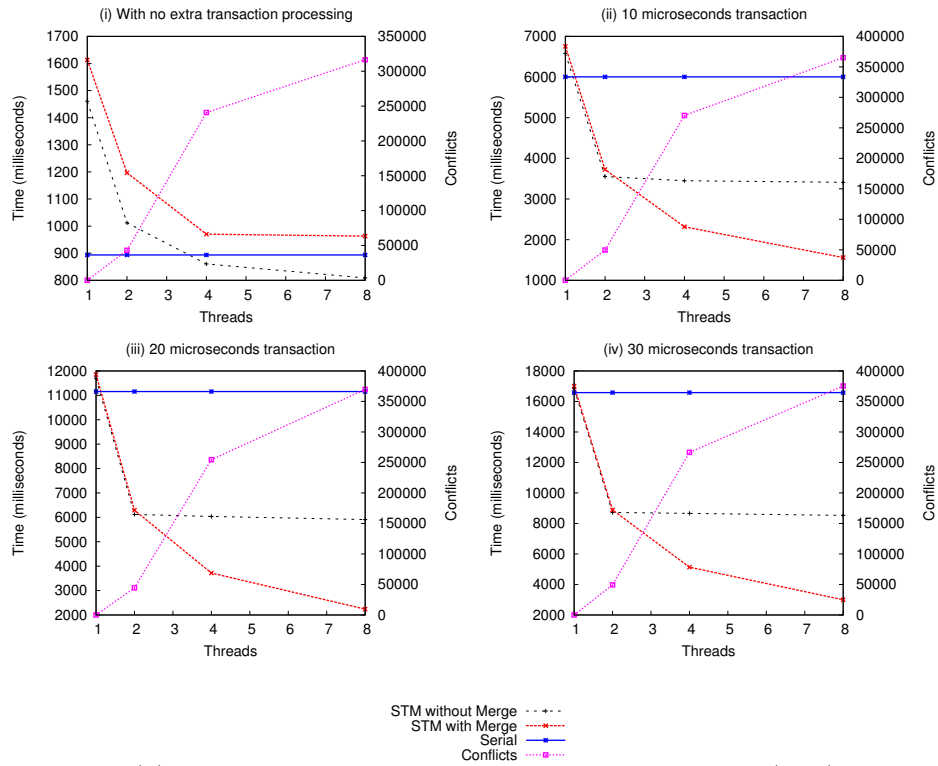
3.7 Summary

While speculation does entail mis-speculating at times, too much mis-speculation can slow down execution. In this chapter we discussed how exposing the semantic meaning of the speculation can minimize the impact of these incorrect speculations by salvaging its work and by *merging* it with another transaction through the *merge* construct. We motivated the *merge* construct and used the connected components problem to explain how it can benefit graph applications that are speculatively parallelized. We address the biggest weaknesses of these types of applications: discarded work and poor scalability. We took an in-depth look at the performance implications using the connected components benchmark and demonstrate very significant speedups in the minimum spanning tree benchmark with sustained scalability. We believe the merge

construct provides a simple yet effective mechanism to improve the performance of certain parallel applications which are speculatively parallelized.



(a) Results of the Connected Components benchmark (DS1)



(b) Results of the Connected Components benchmark (DS2)

Figure 23: Results of the Connected Components benchmark

CHAPTER IV

GUIDING SPECULATIONS USING DATA STRUCTURE SEMANTICS

Irregular algorithms or pointer-based algorithms relying on graphs and trees, are heavily utilized in applications today. Given their reliance on pointers, these algorithms are difficult to analyze and their structure (in terms of memory accesses) is obfuscated. This makes the extraction of parallelism difficult. Hence, speculative parallelization is an attractive option but can be tremendously inaccurate. However, for many of these algorithms, the programmer often knows of access patterns which can be used to guide speculation; but is just limited by his inability to express them. For example, a depth-first traversal of a tree has a well defined access pattern and, given a particular node, the programmer knows which other nodes will be looked at next but is currently limited in how to express this information in a useful manner.

In this work, we present a framework that is capable of reasoning about the *semantics* of the dynamic data-footprints of speculative operations to determine their potential overlap which can be used to guide speculations. This knowledge allows our runtime to make either a parallelization decision or guide and throttle concurrency to improve performance (in STMs in particular). Our framework relies on programmer-supplied predicates that are evaluated appropriately at runtime and utilized to assert certain properties about data-footprints. It is capable of finding parallelism that is difficult to find statically.

We present simple abstractions and a low-overhead runtime to support our framework. We demonstrate our work by parallelizing a graph-coloring benchmark and by improving the transactional performance of benchmarks from the STAMP suite.

4.1 Introduction

We define irregular applications as applications that rely heavily on pointer-based data structures such as graphs or trees (the STAMP benchmarks for example [21] as well as the older Olden benchmarks [22] some of which have been taken up in STAMP). An important characteristic of these applications is that their memory access patterns are heavily data dependent and cannot be known until runtime. This cripples potential static analysis such as those used to efficiently parallelize dense matrix computations which are some of the most useful for HPC.

However recent work by the Galois project [74, 89] has shown that there is significant parallelism potential in irregular applications. Current approaches to exploiting this parallelism are three-fold:

- **Domain specific:** Domain experts may hand-craft new parallel algorithms that clearly expose the available parallelism. This approach, while the most efficient, is also the most difficult as it requires a parallel rethinking of all algorithms and a specific approach taken for a given algorithm will not necessarily translate to other algorithms.
- **Dynamic dependence testing:** Dynamic dependence testing such as Jade [102] rely on programmer supplied ‘access specifications’ to dynamically determine a-priori where parallelism can occur. Before an operation is executed, its access specifications are evaluated and a runtime can detect if the potential accesses of the operation conflict with those of other concurrently running operations. If no conflicts exist, the operation can be launched in parallel.
- **Optimistic execution:** The Galois model, for example, has taken an approach based on the optimistic execution of potentially parallel sections of code. The Galois system provides a library of common data-structures used in irregular

applications and enforces certain atomicity properties that provide for the correct execution of the application. While this approach is straightforward and accessible to a “Joe” programmer, it can incur significant overhead (enforcing atomicity, recovering from atomicity violations, etc.). The Galois model seeks to parallelize a sequential algorithm, however, it is also possible to utilize optimistic execution in parallel programs replacing critical sections with atomic ones (transactional memories).

In this chapter, we present a fourth approach to exploiting parallelism in irregular applications that, like dynamic dependence testing, relies on programmer supplied *computation predicates* but unlike previous approaches, captures the semantic of an operation rather than simply its effect on accessed objects. Our approach allows us to **i**) better parallelize code in a way similar to the Galois model under certain conditions or **ii**) for existing optimistic parallel code, reduce the overheads intrinsic to the method, in effect extracting more performance from STMs.

4.1.1 Key to parallelism: data disjointness

The key to success in both approaches is understanding the data footprint of an operation. In this chapter, we define a computation as being composed of **i**) an operation and **ii**) a data-extent or footprint. Mendez-Lojo et al. understood the importance of this view in [85] where they stress the importance of a *data-centric* view of a computation. They contend that instead of thinking about dependencies between operations, one must take a view that encompasses the actions of the operations on the data. The data-footprint of an operation, which we will note $D_f(O)$ for an operation O encompasses this information and conceptually contains information about all memory accesses of O .

In dense-matrix operations, $D_f(O)$ is easy to compute and reason about (at least in many cases) and this has led to tremendous productivity improvements in the

HPC domain. Indeed, since dense-matrix operations rely on indexing, a compiler can reason about the ranges of these indices and statically determine whether or not parallelism is present. In irregular applications, this is no longer the case as $D_f(O)$ depends, more often than not, on the *runtime* value of variables which are not as easily bound as indices. Furthermore, the use of pointers complicates analysis as the variable from which to derive a value may be indirectly accessible through another variable.

A key contribution of our work is to allow a runtime to reason about $D_f(O)$ in a way similar to how a compiler can reason on array-based data footprints in dense matrix computations.

Disjointedness property It is crucial to note that disjointedness is really a property linked both to the *operation* and the *data* and the way they interact at runtime. For example, consider nodes and edges in a graph. No information can be inferred about the disjointedness of two operations without knowledge of the semantics of the operations themselves. The fact that an ‘edge’ exists between two nodes does not mean that two separate operations each operating on one of the nodes are overlapping. They will be disjoint if, for example, the operations do not refer to the neighbors of the nodes they are dealing with. The presence of a “link” in the data is therefore not an indication of a “link” between the data-footprints. The association of the state of the data with the semantics of the operation is what determines the disjointedness property between two operations.

Our approach In our approach, we introduce two differentiating elements: **i)** we do not necessarily rely on exact and complete knowledge of $D_f(O)$ but rather incrementally improve an approximation of it and **ii)** we compute $D_f(O)$ in terms of semantically significant elements and not necessarily actual objects of bytes that are accessed. For the latter point, consider an operation that explores a planar space

by picking a point at random and exploring a square around it (for example, a local search algorithm). This type of algorithm is highly irregular as it is unclear what points in the plane will be touched when. A traditional construction of the footprint of the operation would add all the points that are potentially touched. However, it is clear that simply storing a tuple containing the center point and the size of the square contains all the information required to determine if two operations are disjoint. Our approach therefore places a strong emphasis on the link between elements in $D_f(O)$ and the semantics of O . We provide programmer APIs to specify this link. We believe that moving away from a purely memory oriented view of an operation’s footprint is key in allowing the simple expression of complex access patterns.

Furthermore, we incrementally improve the completeness of $D_f(O)$ instead of always requiring precise knowledge of it. We will see that this is useful in improving STM performance where completeness is not a requirement. For the purpose of this chapter, we will consider operations O_1 and O_2 in parallel with $D_f(O_1)$ and $D_f(O_2)$ their respective data-footprints. We further define *inputs* to an operation as any value that is known at the point of launch of an operation. More precisely, we seek to determine approximations of $D_f(O_1)$ and $D_f(O_2)$ given I_1 the input to O_1 and I_2 the input to O_2 . These approximations will then allow us to make assertions on the disjointedness of the two footprints and therefore determine whether or not two operations are parallelizable.

Static approach Unlike other methods (Dynamic Parallel Java (DPJ) [18] for example), we do not provide static compiler-checkable assertions about the footprint of an operation. In fact, our definition of footprint allows for it to be incomplete. DPJ is a language that provides a type and effect system that is verifiable at compile time allowing the compiler to make strong assertions about the effects of a particular

operation on the data. The strong guarantees that DPJ makes about a program allow it to make some very interesting static optimizations like eliminating some of the concurrent overhead of locks or transactions when it can be statically certain that no conflict is possible. In our approach, while we do not provide the convenience and security of a statically deterministic execution, we allow the programmer to capture cases where the presence of parallelism is dependent on the *value* of a variable as opposed to its type (or even extended type as in DPJ). Our system is thus complementary to static approaches and can be used to further improve performance. The cost of the flexibility though is the loss of a static guarantee of the execution.

Dynamic parallelism approach Another approach that has been explored is the dynamic detection of parallelism such as Rinard’s Jade [102]. However, whereas Jade focuses on identifying conflicts at a byte level, we utilize semantic information and the programmer’s knowledge about an operation to determine whether or not two operations overlap.

4.1.2 Motivating example

Despite their *irregularity* due to their pointer intensive code, irregular applications do contain structure and their data-access patterns are frequently well-known to the programmer. Take for example the simple greedy graph coloring algorithm given in Figure 24(a) as pseudo-code.

From the code, it is clear that the data-footprint of an iteration of the loop on Line 4 is composed of the node `curNode` on Line 6, its neighbors in the graph and `lastColor`. This information is easily obtainable given only `curNode` which is known at the beginning of the iteration. However, this information is not statically known nor can it be reasoned about at compile time with a type and effect system because the exact “neighbors” of a node cannot be known at compile time. The disjointedness property is dependent on the runtime values of the adjacency matrix for example (if

```

1 Graph g;
  List vertices;
  int lastColor = MAX_COLOR;
  while(!vertices.empty()) {
    Set neighborColors;
6   Node curNode = vertices.pop_head();
    Vector neighbors = curNode.getNeighbors();
    for(int i=0; i<neighbors.size(); ++i) {
      if(neighbors[i].isColored()) {
        neighborColors.push_back(
11         neighbors[i].getColor());
      }
    }
    if(neighborColors.size() >= lastColor) {
      printf("Failed to color\n");
16  } else {
    for(int i=0; i<lastColor; i++) {
      if(neighborColors.find(i))
        continue;
      curNode.setColor(i);
21  }
    }
  }

```

(a) Greedy graph coloring algorithm

```

1 DEFINE_VALUETAG(NODE_T, Node);
  DEFINE_OPERATORROLE(COLOR_F);
  Graph g;
  List vertices;
  int lastColor = MAX_COLOR;
6  while(!vertices.empty()) {
    Set neighborColors;
    Node curNode<NODE_T> = vertices.pop_head();
    OPERATION(COLOR_F, curNode) {
      Vector neighbors = curNode.getNeighbors();
11      /* Original code is unchanged here */
    }
  }
  DISJOINT(NODE_T v1, COLOR_F,
16  NODE_T v2, COLOR_F) {
    if(g.hasEdge(v1, v2)) {
      return OVERLAP;
    }
    return DISJOINT;
21 }

```

(b) Greedy graph coloring algorithm using our abstractions (Section 4.3 has the details)

Figure 24: Motivating example: a greedy graph coloring algorithm is shown in 24(a) and a modified version using our abstractions is shown in 24(b).

edges are noted through an adjacency matrix).

The combination of the *semantics* of the operation (“coloring”) and the values of an underlying data-structure (the adjacency matrix) form the basis for the decisions on disjointedness. A different operation with different semantics but on the same data-structure would produce different disjointedness properties. For example, one could imagine a modified graph coloring algorithm that assigns a color not used by any neighbors and their neighbors. Although the data-structure would be the same, the semantics of the operation would change the disjointedness predicate.

In this chapter, we seek to define *predicates* on operations and their inputs that allow assertions to be made about the data footprint. In the example in Figure 24(a), the properties of the footprint are nicely captured in a simple edge presence test. However, this may not be the case in all applications and we will define and justify general properties about $D_f(O)$ that are true for a wide range of irregular applications.

4.1.3 Use-cases for $D_f(O)$

We propose using semantic information about data footprints in two distinct contexts: **i)** dynamically extracting parallelism and **ii)** improving the performance of optimistic parallelism through a concurrency scaling mechanism. In the first case, the programmer writes a sequential application and parallelism is dynamically extracted where possible. In the second case, the original application is already parallelized using optimistic techniques (such as STMs) and we improve performance.

Parallelization In many parallel models, a `foreach` construct exists where each iteration of the loop will execute in parallel. This construct, present in the Galois model as well as the DPJ model, is very similar to the `DOALL` construct in FORTRAN except that the iterations are optimistically executed in the case of the Galois model and statically checked for overlaps in the case of the DPJ model.

In our approach, we use the dynamic semantic knowledge of $D_f(O)$ to determine whether or not an iteration O should be executed in parallel with the other iterations (if they are disjoint) or launched at a later point. When launched in parallel, no locks or any other overhead incurring mechanisms are used as our framework guarantees that the operations will not conflict. Note that in this case, the requirement is that $D_f(O)$ be fully known and not just an approximation. Our approach can be viewed as a middle road between a fully static parallelization approach and a fully optimistic approach (such as STM).

Throttling concurrency For applications that already make use of concurrency optimistically we propose a method of *throttling* the start of transactions based on their approximate data-footprint. The goal is to reduce the number of aborts of these transactions by being fairly certain that no conflict will occur. Note that contrary to the first case where we required certainty of non-overlap to launch without any

concurrency checks, here we maintain the STM semantics and simply throttle the start of a transaction to try and maximize the likelihood of it committing successfully. Here, our support for incrementally better approximations takes its full importance: we can trade-off confidence in disjointedness with the time and complexity required to build the full footprint.

As our approach focuses specifically on reducing the number of aborts, it is particularly well suited to long running transactions. This is of particular interest as long running transactions are usually avoided in STM systems due to their high cost in case of failure.

The remainder of this chapter is organized as follows. Section 4.2 lays out the two key predicates in our framework. Section 4.3 describes our runtime implementation and how best to utilize the information provided through the predicates. Section 4.4 showcases our experimental results. Section 4.5 presents the related work and we summarize in Section 4.6.

4.2 Expressing data-structure semantics

Given an operation O_1 operating on an input I_1 in an irregular application, we seek to determine certain general properties on $D_f(O_1)$ using only the knowledge available from the runtime value of I_1 at the start of O_1 .

4.2.1 Properties on the data-footprint

Analyzing in a meaningful manner $D_f(O_1)$ and $D_f(O_2)$ is only a means to an end; ultimately, the goal is to be able to determine whether O_1 and O_2 can execute in parallel or not, in other words if they have an overlapping memory footprint. Initially, all that is known is that $\{I_1\} \subseteq D_f(O_1)$ and $\{I_2\} \subseteq D_f(O_2)$.

We define two predicates that can be used to determine $D_f(O_1)$ (and similarly $D_f(O_2)$):

- **Disjointedness** For any given pair of objects in the space $(D_f(O_1) \times D_f(O_2))$, we can determine whether or not the two objects will entail accessing overlapping areas given the current operations they are in. This predicate expresses the programmer's *semantic knowledge* of how objects in the footprint relate to the operations operating on them. In the example of the local search in a plane, the disjointedness predicate would express the fact that tuple (C_1, R_1) and tuple (C_2, R_2) are disjoint if and only if $\text{dist}(C_1, C_2) > R_1 + R_2$. This condition is only derivable from the semantic knowledge of how the operation acts on the tuples.
- **Growth** For any object in $D_f(O_1)$, we can determine the *next* objects that are semantically important to consider in the given operation. The growth operation allows the successively better approximations of $D_f(O_1)$ to be constructed from one another.

The specifics of each predicate are given in the following subsections but it is important to understand how both of these predicates serve to give meaningful information about $D_f(O_1)$. Initially $D_f(O_1)$ is only composed of a single element which is the input passed to the operation. We can therefore deduce the crudest possible *approximation* of the data-footprint. By adding elements to D_f by determining the important elements that will be accessed due to accessing I_1 (for example, neighbor nodes in a graph), we can *grow* D_f to be able to deduce a better approximation of the memory footprint. While this may seem like a very expensive operation, in practice it is actually relatively cheap and can be performed at a fraction of the cost of an otherwise long running operation. Note also that the approximation can be terminated at any time; while less precise than the full approximation, it might be sufficient to make a determination about two concurrent transactions.

In the example given in Figure 24(a) for example, the programmer does not have

to grow the footprint to include all the neighbors but can quickly make a semantic assertion about the disjointedness of an element with respect to another. The true power of our framework lies in being able to make these assertions as they allow the programmer to rise above a purely memory-oriented data-model to a more semantics-oriented model (ie: locations in memory are not important, rather it is what the value is going to cause that is important).

Disjointedness predicate Going back to the example in Figure 24(a), to determine whether two iterations of the main loop have a disjoint footprint, the programmer does not need to actually construct the footprints but simply determine if there is an edge between the two nodes given as input to the iterations. This is therefore an example where knowledge of only the inputs (I_1 and I_2) allows us to make a disjointedness claim about the data-footprints.

The disjointedness predicate is defined for pairs of elements. Note that the operations in which the elements are being used are also important. Depending on the value of the elements, the semantics of the operations using them and the state of the computation, the predicate can determine whether or not the two elements are disjoint (ie: operating on them will not produce an overlap). In the graph coloring example, the two elements are the two nodes to color and the “coloring” operation is applied to both of them. We define the disjointedness property as follows:

Definition Let O_1 and O_2 be two distinct operations, for any element a_i in $D_f(O_1)$ and any element b_j in $D_f(O_2)$, we define the pair-wise disjointedness of (O_1, a_i) and (O_2, b_j) as a function that returns whether the two elements will result in accesses to disjoint memory locations, overlapping memory locations or an unknown memory locations (and therefore, nothing can be said about disjointedness one way or the other).

$D_f(O_1)$ and $D_f(O_2)$ will overlap if and only if at least one of the a_i s overlaps

with at least one of the b_i s. We will say that $D_f(O_1)$ and $D_f(O_2)$ are fully disjoint if every pair is disjoint.

We will refer to this predicate as $\text{Pred}_D(O_1, a_i, O_2, b_j)$.

Growth predicate In the example of graph coloring, disjointedness could be determined solely based on the values I_1 and I_2 , that is, the initial known subsets of $D_f(O_1)$ and $D_f(O_2)$. However, it is not always possible to determine disjointedness based on so little information. Take for example a search algorithm that explores a graph until it finds a specific value. Given two starting points, it is difficult to determine the disjointedness of the two footprints. However, the programmer, even if he does not know whether the two footprints are disjoint can *constructively* expand the footprints. In our example, all the neighbors in the graph of the starting point can be added to the footprint. This process can be recursively repeated until one of the neighbors is the node that is being searched for.

Note that in this case, the construction of the full footprint takes just as long as the execution of the algorithm itself and is therefore not very useful. However, since the process can stop at any point, more and more precise approximations may be built at a fraction of the cost required to build the full footprint.

We will refer to this predicate as $\text{Pred}_G(O_1, a_i)$.

Two useful predicates Individually, the Pred_D and Pred_G are useful in collecting information about the data-footprint of the operations. They can also be combined to gain even more insight on the footprint. Indeed, the disjointedness predicate is a *per element pair* property and can therefore be evaluated for all the new elements discovered through the growth predicate. The discovery of an overlapping pair will mean that the data-footprints are overlapping and that therefore the operations relying on them cannot run in parallel. The quicker this assertion can be made, the more efficient our system will be.

The alternating application of both predicates thus allows deeper and deeper knowledge to be gained about the footprints of the operations. This is a potentially exponential process but in practical and useful cases, we have seen that it terminates very early. We will demonstrate this in Section 4.4.

4.2.2 Specification

In this Section we define the concepts needed to allow the programmer to utilize the two predicates introduced above: `OperationRole` and `ValueTag`. With these concepts, the programmer can identify the *semantic type* of functions (`OperationRole`) and data (`ValueTag`). This is crucial as the disjointedness predicate links both of these concepts in ways that are specific to each `OperationRole` and `ValueTag`.

OperationRole: Encapsulating operation semantics An `OperationRole` is an identifier given to a set of operation semantics such as “traverse depth-first”, or “traverse breadth-first” or even simply “traverse”. An `OperationRole` can be viewed as a tag that identifies the current function to the runtime system. Different functions may have the same `OperationRole` if they have the same semantic role.

ValueTag: A semantic tagging system Similarly to the `OperationRole`, values also play a “role” in the predicates: an operation may act differently on the initial node and its children node. The C/C++ type system is not concerned with distinctions based on the role of a value in the program and is therefore inadequate to capture the semantic meaning of a value. Furthermore, dynamically, the role of the same value may change and a static type system cannot capture this. We therefore propose the `ValueTag` dynamic tagging of values. The tagging allows the runtime to determine what part the value plays in a computation.

Consider again the example in Figure 24(a). The original node used as input

and the neighbor nodes used inside the computation play different parts in the determination of the predicates. Indeed, only the original node is used to determine disjointedness with other operations whereas Pred_D is not called for any of the children nodes. As far as C/C++ is concerned, they are all pointers but they have a semantic meaning associated with the computation. This semantic meaning can change as time progresses and a tag instead of a type addresses this issues. Therefore, for a single C/C++ type, multiple “roles” may exist and this is captured by the `ValueTag`. Conversely, a single `ValueTag` may also encompass different C/C++ types.

4.2.2.1 Formal definition of the predicates

With the two concepts in hand, we can formalize the definition of Pred_D and Pred_G .

Definition We define \mathcal{O} as the space of `OperationRole` and \mathcal{V} as the space of `ValueTag`. In practice, they are finite sets of elements.

Definition of Pred_D We define \mathcal{D} as the set of the three elements `DISJOINT`, `OVERLAP` and `UNKNOWN`. \mathcal{D} represents the result of Pred_D . Pred_D follows the prototype given in Equation 1.

$$(\mathcal{O}, \mathcal{V})^2 \longrightarrow \mathcal{D} \quad (1)$$

It respects the following properties:

- The result of Pred_D is `DISJOINT` if and only if the input `ValueTags` used in their respective `OperationRoles` will not result in accesses that will conflict (in other words, no write and read or write and write to the same location).
- Conversely, the result of Pred_D is `OVERLAP` if and only if there will be a conflict.

- In all other cases, the result is UNKNOWN and no assumption can be made about the disjointedness of the elements.
- The function is symmetric and $\text{Pred}_D(O_1, a_1, O_1, a_1) = \text{OVERLAP}$

Definition of Pred_G Pred_G follows the prototype given in Equation 2 where n is an integer that indicates the number of values that will be accessed next through the input value.

$$(\mathcal{O}, \mathcal{V}) \longrightarrow \mathcal{V}^n \tag{2}$$

Note that the result encompasses a dynamic `ValueTag` but also the actual runtime value of the variable returned.

4.3 *Runtime usage and implementation*

In Section 4.2 we formally defined Pred_D and Pred_G . In this Section, we describe how these concepts translate into C/C++ code and how the runtime makes use of them during the execution of the program. We implemented our work in C++ due to the wide availability of accepted benchmarks in C and the power of template meta-programming in C++ (crucial to an efficient implementation of our runtime). Most of the visible API is defined as macros that hide the actual complexity of template meta-programming.

We also describe the role of the runtime in efficiently making use of the predicate information. Indeed, we saw that potentially Pred_G can return a huge amount of data which would require a long time to check instead of doing useful computation. Finally, we present how the runtime improves both parallelization and successfully throttles transactions to lower their abort rate and improve their execution time.

Note that given our two goals (parallelization and optimization of transactional code), we have slightly different runtimes for both. However, the concepts are similar

and unless otherwise noted, the concepts described here apply to both applications.

4.3.1 Programmer specifications

The programmer first needs to identify the `ValueTags` and `OperationRoles`. They are just labels and, while they carry semantic meaning for the programmer, they carry no particular meaning for the runtime except that it can compare them (test for equality). We will use the example in Figure 24(b) as a running example of the requirements from the programmer. The code given closely reflects our actual implementation which shows that the burden on the programmer is very light. Indeed, the actual implementation uses simple macros to wrap more complex template syntax.

Specifying \mathcal{O} and \mathcal{V} The programmer is responsible for enumerating the semantic types he is interested in tracking. This task is relatively simple and follows from the design of the algorithm itself. Each operation can be statically annotated by its `OperationRole` and each variable can be dynamically tagged (as in, the tag may depend on the control flow) by its `ValueTag`. We note that `OperationRoles` are considered to be scoped with the most deeply nested one the active one. In Figure 24(b) the `OperationRole` is defined on Line 2. In this simple example there is only one `OperationRole` but there is no fundamental reason why this should always be the case. The `ValueTag` is identified on Line 1. We note that a `ValueTag` is associated with a single C/C++ type. The reverse is not required though as the same variable may be tagged with different `ValueTags` at different times in the program. The restriction to a single C/C++ type for a `ValueTag` is due to an implementation detail but does not harm functionality in practice.

We see how the `curNode` variable is tagged with the `NODE.T` `ValueTag` on Line 8. Note that although this seems static, the framework allows the dynamic tagging of variables, in other words, the exact tag may depend on the control flow of the program. This indicates to the runtime that `curNode` has a value type of `NODE.T` and

this will imply certain predicates that can apply to it.

Specifying Pred_D and Pred_G Now that the user has defined \mathcal{O} and \mathcal{V} , the predicate functions are simply overloaded functions that are distinguished based on their input `OperatorRoles` and `ValueTags`. The programmer does not need to define all possible combinations as Pred_D and Pred_G are in essence overloaded functions, one for each distinct combination of parameters from \mathcal{O} and \mathcal{V} . The programmer can therefore specify only those that make sense and the correct function will be selected at runtime. In our example, this is defined on Line 16. We note that we only define Pred_D which determines whether or not an edge exists between the two `NODE_T` (called `v1` and `v2` in the code).

Specifying the operation The programmer has now specified the `OperationRole` and `ValueTag` possibilities as well as the predicates that make sense; the only remaining information the programmer needs to provide is the identification of the operation itself. This is shown on Line 9. The `OperationRole` is defined as well as its initial input. This specification will bind the predicates that are applicable to the `OperationRole` of the operation (here `COLOR_F`) and the `ValueTag` of the input variable (here `NODE_T`). The extent of the operation is also indicated with the curly braces and the code within the operation does not need to change as it will either be launched with no concurrency overhead or sequentially in case a conflict is detected.

Summary The programmer does not need to supply any more information. To review, the programmer is required to define **i)** the space of the `OperatorRoles` and the `ValueTags`, **ii)** the predicates, **iii)** the `ValueTag` for variables that are used as input to operations and **iv)** the operations themselves.

Provided the programmer understands the semantics of his program, these requirements are easily met. It is possible that some programs are not amenable to

the predicates we define but in our experiments, we have found that many of the benchmarks were amenable to such transformations and although our framework did not always provide a significant benefit, the work required from the programmer was minimal.

4.3.2 Low-overhead runtime

We have implemented our runtime in C++ and made heavy use of templating mechanisms to allow the selection of the appropriate predicate function for all combinations of values from \mathcal{O} and \mathcal{V} . This significantly improves the runtime binding operations for these predicate functions.

The main role of the runtime is to apply the appropriate predicates when an operation is being launched and to determine whether or not it can concurrently execute with the other currently executing operations. The runtime will follow the algorithm shown in Figure 25 to best approximate the data-footprint of the operation.

4.3.2.1 Degree of approximation

However, the runtime will also estimate how long it should take for the checks. This is particularly important if there are many concurrent operations as a new incoming operation has to be tested against all existing operations. The intuition behind measuring runtime overhead is that we do not want the time required to check for disjointedness to be significantly more than the time we would waste by not running in parallel or by running in parallel and having to abort (in a STM system). Therefore, the runtime maintains statistics about the execution of the operations and uses them to determine how much time it can use up to check for disjointedness. It will then step out of the algorithm shown in Figure 25 whenever it has spent at least that much time performing checks.

```

Input: initialElement the initial element passed to the operation
Input: currentOperation the operation that is starting
Input: currentFootprint  $D_f$  (currentOperation) (being constructed)
Output: overlapStatus whether or not the operation will conflict (to the best
of the runtime's knowledge)
isUnk  $\leftarrow$  false ;
currentFootprint  $\leftarrow$  initialElement while !currentFootprint.empty() do
  currentElement  $\leftarrow$  currentFootprint.pop_front() ;
  foreach ConcurrentOperation concOp do
    foreach ElementInFootprint existingElt do
      res  $\leftarrow$  isdisjoint(currentOperation, currentElement, concOp,
existingElt) ;
      if res == OVERLAP then
        overlapStatus  $\leftarrow$  OVERLAP ;
        return overlapStatus;
      end
      if res == UNKNOWN then
        isUnk  $\leftarrow$  true ;
      end
    end
  end
  foreach elt in Grow (currentElement) do
    currentFootprint.append(elt) ;
  end
end
if isUnk then
  overlapStatus  $\leftarrow$  UNKNOWN ;
else
  overlapStatus  $\leftarrow$  DISJOINT ;
end
return overlapStatus;

```

Figure 25: Simplified runtime algorithm to determine whether `currentOperation` can execute in parallel with the existing operations.

Monitoring for the parallelization approach In the simplest case, the runtime has the choice between running an operation in parallel with no concurrency checks or running sequentially (in case of a conflict). Here, the time that the runtime keeps track of is the average time required for an operation. Indeed, that is the “cost” of not running it in parallel (as it will have to be serialized). The runtime will then perform checks for a small fraction F of that time to decide whether or not to launch it in parallel.

Monitoring for the throttling approach In this case, the “cost” of a bad check is a rollback. The cost of the rollback includes the cost to partially execute the transaction (an operation here) and the overhead of the actual rollback mechanism.

However, a rollback does not always occur so on average, the cost incurred is $\frac{T_A}{C_C+C_A}$ ¹ where T_A is the total amount of time that the transaction spent aborting, C_C is the commit count and C_A is the abort count. This quantity captures both the likelihood of a rollback and the average cost of that rollback. We can see that if no rollback has ever occurred, the runtime will not perform any checks and just let the transaction run as it is highly likely that there will not be any conflict (given past history). Again, the runtime will perform checks for a fraction F of this quantity. F is a user-defined fraction.

4.3.3 Runtime usage

We set out to **i)** improve parallelization and **ii)** improve the performance of transactional systems through the throttling of their transaction. This Section explains how the runtime is used to implement these goals

Improving parallelization In this context, the runtime has to be positive that disjointedness is maintained since the goal is to launch the operations in parallel with no runtime checks (such as STMs). We make the assumption that the programmer provides accurate and complete predicates. Under that assumption, if $D_f(O_1)$ and $D_f(O_2)$ are *fully disjoint* (ie: all the pairs are disjoint) and no elements can grow the data-footprint, it is safe to run both O_1 and O_2 in parallel. This condition is very constraining and may not be possible to check in a small amount of time but in the graph coloring example which was our motivating example and which we demonstrate in our Section 4.4, this condition is trivial to verify and provides good results.

If the runtime check cannot conclusively determine disjointedness, the safe option is chosen and the operation will be serialized. Note that since the time of the runtime check is limited to a small user-defined fraction F of the average sequential time of

¹We assume here that other transactions could have run had the transaction that rolls back had not run.

the operation, in the worse case where no parallelism can be safely found, we incur a slowdown over the sequential execution that is completely determined by F .

Throttling transactions In this case we do not need to gain as much certainty about the disjointedness of two operations as we are still protected by the STM system. We simply seek to reduce the number of aborts. We therefore perform the same checks as for the previous within the time frame allotted. If we discover an overlap, we pause the transaction before its launch to allow the conflicting transaction to commit first. Similarly, if we determine that operations are fully disjoint, we allow them to proceed. In the case where a determination cannot be made, we also allow them to proceed and let the STM system take care of conflicts. The longer the check is allowed to go and the more certainty is gained about the disjointedness of two operations. The fraction F again determines the level of trade-off between certainty and performance the programmer is willing to accept.

4.4 *Experimental evaluation*

In this section we demonstrate the benefits of our approach through a simple greedy graph coloring algorithm as well as through several STAMP benchmarks to illustrate its wide applicability.

All experiments were performed on a dual quad-core Intel Xeon E5540 (2.53GHz) with up to 8 concurrent threads. We chose not to increase threads past this limit to free ourselves from the issues related to kernel level thread scheduling

4.4.1 Greedy graph coloring

The greedy graph coloring algorithm was introduced in Section 4.1.2. In Section 4.2 we explained the applicability of the two predicates Pred_D and Pred_G . We also discussed how it is extremely simple for the programmer to specify a Pred_D predicate which determines if $D_f(O_1)$ and $D_f(O_2)$ are disjoint. All the predicate needs to do

is determine if there is an edge between two inputs I_1 and I_2 .

We implemented two versions of the algorithm (Figure 24) in C++.

- The first was a traditional parallel version of the algorithm. The processing of each node was assigned to a different task. We used an STM (TinySTM [121]) to wrap the critical section using atomics. We used the `parallel_for` construct in TBB [?] to launch all of our tasks in parallel. TBB also managed the mappings of tasks to the underlying threads. This was the baseline.
- The second was a parallel version which applied our approach. This version created a number of TBB tasks and scheduled them based on the execution of the Pred_D predicate. Instead of blindly selecting the next task from the run queue, the scheduler postponed the execution of any task which was not disjoint with the other tasks that were currently running (through the evaluation of the Pred_D predicate). Indeed, we could have waited until the conflicting task could be scheduled again and spun in a tight loop until that time. But, by postponing the execution and choosing another disjoint task allows us to make much more use of the available parallelism. In this version, we were able to avoid the use of any locks or STMs around our data structures, since we guaranteed that the execution of any two concurrent tasks would not have an overlapping data footprint.

The input dataset to the two versions consisted of a randomly generated graph which consisted of 2000 nodes with an average out-degree of 80.

We note that our approach is particularly beneficial for long running transactions. As discussed before, this is of particular interest as long running transactions are usually avoided in STM systems due to their high cost in case of failure. To study the behavior of our system in the presence of long running transactions we simulated transactions of different durations and observed that as the duration of the

transaction increased the speedup achieved increases. This is expected, since as the duration of a transaction increases, its associated cost of rollback (in the case of a conflict) correspondingly increases. Hence, assurance of no conflicts before launching a transaction dramatically increases performance.

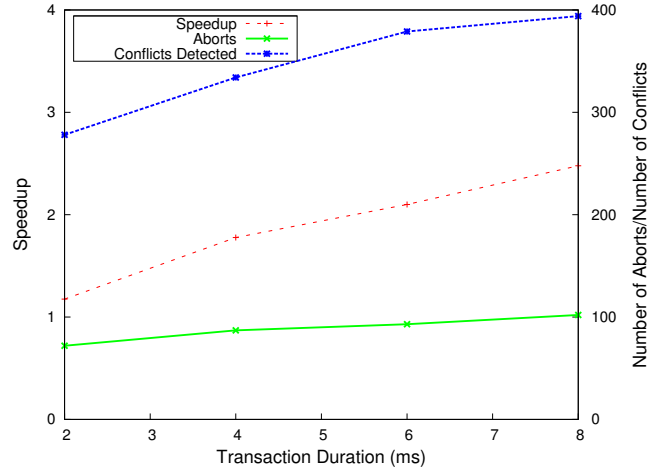


Figure 26: The Greedy Graph Coloring Benchmark

Figure 26 shows the performance of our system with transactions of varying durations. The speedup line indicates the ratio of time it takes to execute a traditional parallel implementation to an implementation which uses our approach (in other words, this is not the speedup relative to a sequential implementation but rather expresses the benefit of our method over a traditional parallel implementation). The “Number of aborts” is the number of aborts that the STM system had to perform (in the case of the traditional parallel version). The “Number of conflicts” is the number of times our scheduler detected the a conflict in the data footprint through the execution of the Pred_D predicate and decided to postpone the execution of the task (in the case of our enhanced parallel version).

We observed that with a transaction duration of 8ms the system was able to provide a speedup of over 200% over a traditional parallel implementation. The “Number of conflicts” increases as we increase the duration of the transaction since there is more potential for conflict. This number indicates the number of times our

scheduler postponed a task to prevent the possibility of a conflict. The “Number of aborts” increases as the number of threads increase as expected. This is because the amount of contention increases and the STM system needs to rollback more frequently. Note, that the total cost of aborting is proportional to the number of aborts and the duration of a transaction.

4.4.2 STAMP benchmarks

We also modified several STAMP benchmarks to take advantage of our approach by adding simple predicates. In particular, we modified the KMeans (K-means clustering), the Yada (Delaunay mesh refinement; Ruppert’s algorithm) and the Labyrinth (maze routing) benchmarks. All experiments were performed on a 8-core machine with 8 concurrent threads.

4.4.2.1 *K-Means*

Transactions in the K-Means benchmark wrote into a shared array. Conflicts arose due to the fact that several transactions could potentially be writing into the same parts of the array at the same time. An STM system rolls back transactions in the case of such a conflict. We added an extremely simple Pred_D predicate which determines if two transactions will be accessing disjoint parts of the array or not. The predicate is simply an equality comparison of the indices that each of the transactions will be accessing. We would like to emphasize, again, on how simple it is to write this Pred_D predicate (just one line of code in this case). We were able to achieve a drastic reduction in the number of aborts and the corresponding reduction in execution time through the addition of this code.

4.4.2.2 *Yada*

Transactions in the Yada benchmark (based on the Delaunay thread refinement algorithm) try to “refine” elements by working in a cavity (a neighborhood) around

themselves. Each refinement reads and writes to that cavity and if two elements being processed concurrently are too close to each other a rollback will occur. We therefore added a simple Pred_D predicate which compares the distance from the center point of the elements to the sum of their radius. If the distance between the two elements is larger than a small multiple of the sum of their radius, we consider the transactions to be disjoint and allow them to proceed. In the other case, we halt one transaction to give the other time to finish and avoid an expensive rollback.

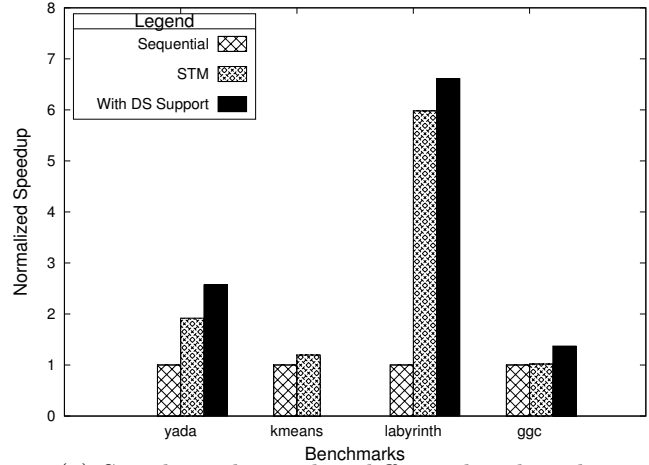
4.4.2.3 *Labyrinth*

In the labyrinth benchmark, different transactions use Lee’s routing algorithm to find the shortest path between a pair of nodes on a 3D grid. Once the shortest path is found, the transactions write back the path to the grid thereby causing a conflict if a concurrent transaction wants to write to the same cell in the grid. To reduce the number of aborts and improve the execution time, we implemented a very conservative Pred_D predicate that compared the spanning cube of the source and destination points of the transactions. If the cubes overlapped, we considered that the transactions had a potential to overlap, otherwise they did not and we allowed them forward. Note that this approach is extremely conservative and could potentially lead to long waits if the cubes are too big. We found however that for more numerous and smaller paths, our approach worked very well.

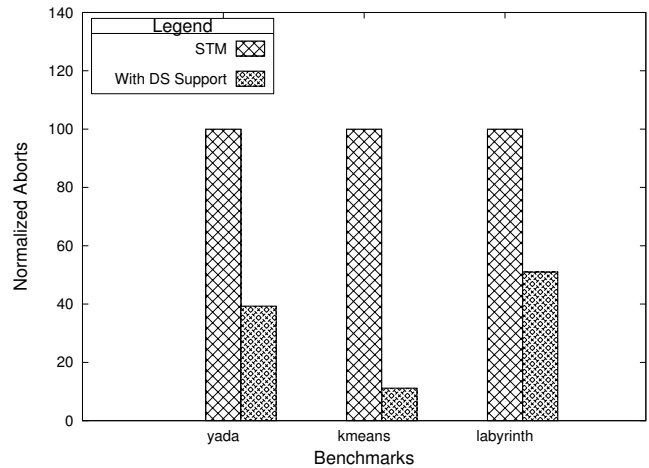
4.4.2.4 *Results*

Figure 27 reports the speedups we were able to achieve as well as the reduction in the number of aborts. Speedups are relative to a sequential execution of the benchmarks.

We note that in all cases the number of aborts drops significantly and the performance increases by a lesser margin but still increases. Note that the fact that aborts still exists is most likely due to hashing conflicts in the lock tables of the STM system. The fact that the number of aborts goes down so dramatically makes our



(a) Speedups obtained in different benchmarks



(b) Normalized number of aborts in different benchmarks

Figure 27: STAMP Benchmarks

system particularly well suited for long running transactions where the cost of an abort is high (in particular for transactional systems that use commit time locking).

4.4.3 Scaling

As the number of threads increase the contention for shared resources in any application typically increases. We studies the scaling properties of our system with the increase in the number of threads. Intuitively, our system should provide more potential for speedup as the number of threads increase, since a traditional STM system would incur the cost of additional rollbacks (with increase in contention).

We performed a more detailed analysis of the Labyrinth benchmark and ran it

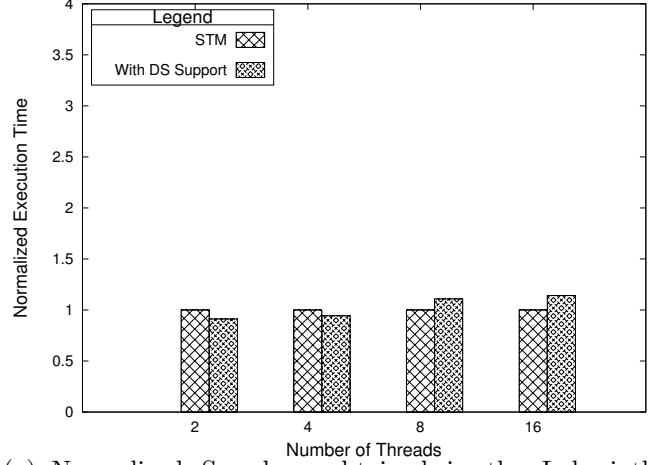
with 2, 4, , and 16 threads. We observed that when the number of threads is low (2 - 4) a slowdown occurs, primarily due to the low amount of contention experienced and the overhead incurred by our system. However, as the number of threads were increased, contention increases and the cost of rollbacks increase. Our system reduces the number of rollbacks dramatically thus leading to effective speedups.

We would like to point out that as the number of threads increase our system provides excellent scalability a compared to traditional approaches and such an approach is essential in the presence of a larger number of concurrent threads. Figure 28 reports the results we obtained my running the Labyrinth benchmark with a varying number of threads.

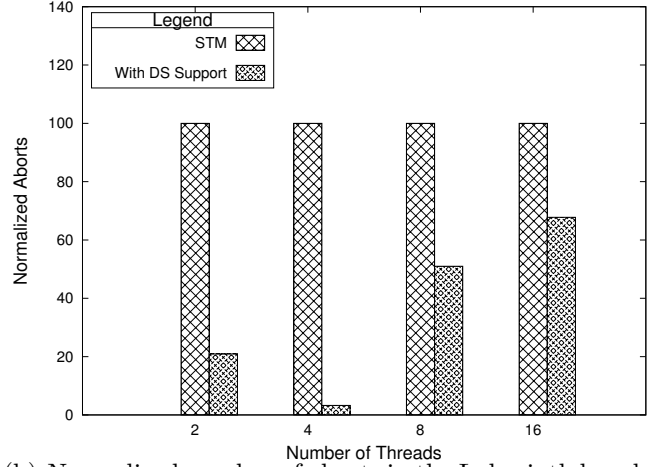
4.5 *Related work*

As previously mentioned, our work is related to that on the Galois programming model [74, 85] as they also take a *data-centric* approach to parallelism. However, their model is based purely on optimistic execution. In [85], they do describe some optimizations to improve the performance of the optimistic system but this is based mostly on refactoring the data-access patterns to make them more STM-friendly. In contrast, our work tries to establish properties of the data-footprint of operations by dynamically evaluating the appropriate programmer-supplied predicates.

DPJ [18] is also similar to our work as it makes assertions about the data-footprint of operations at compile-time. As previously mentioned, we believe this work is orthogonal to ours as we relax the guarantees that we make to the programmer but enable the detection of more potential parallelism. It would also be interesting to combine both frameworks to provide some static assertions and allow the knowledge gain at compile time to aid in the dynamic runtime predicate evaluation. Work by Reid et al. [100] is similar work which also extends the type system by using a notion of static ownership of data. Each data object is *owned* by a logical owner (a parent



(a) Normalized Speedups obtained in the Labyrinth benchmark



(b) Normalized number of aborts in the Labyrinth benchmark

Figure 28: Labyrinth Benchmark

class, a global owner known as “world” for global objects or any other object in the scope) and an effect system is also introduced for functions. Static reasoning similar to DPJ can then be used to perform automatic parallelization.

Automatic dynamic parallelization work such as Rinard’s Jade [102] is closely related to our work in the sense that it also dynamically annotates data structures and dynamically evaluates conditions to determine the potential overlap of operations. Like our approach, this evaluate also happens just in time providing increased flexibility over static approaches. However, Jade evaluates these conditions based on the *memory footprint* whereas we have focused on injecting semantic knowledge about

the operation into the evaluation of conditions. Work on serializer sets by Allen et al. [5] also seeks to dynamically extract parallelism by having the user specify *serializers* which correspond to operations that conflict with one another and therefore must be executed sequentially. This work is interesting because it also brings in some semantic knowledge about the operation to build the serializers however the approach taken is very different from ours.

A lot of work has gone into STMs, in particular to reduce the overheads of the STM runtime. Some recent work [123] attempts to reduce the overhead of STMs by replacing atomic sections by locked sections when appropriate. Many other techniques have also been developed (see [76] for a recent list). However, our work does not specifically target STMs as such but rather STMs can benefit from our analysis.

4.6 Summary

In this chapter we investigated how the semantic data footprints of speculations can be leveraged to guide and throttle speculations as well as provide parallelization opportunities. We discussed how the disjointedness of two operations is really an intricate dynamic relationship between the operation and the runtime values of the data. Therefore, static approaches will not always be able to capture all the expressible parallelism in a program. Our approach uses speculative parallelism to extract parallelism from these applications. We allow the programmer to define some very simple semantic predicates that will get executed at runtime to determine the disjointedness of two operations. Communicating this very simple knowledge that the programmer has about his algorithm can be used to guide speculation and shows promising results. We have shown that we are able to parallelize certain sequential algorithms in an efficient manner without incurring the overheads of a transactional system. For algorithms utilizing a transactional system, we have shown that we are able to improve their performance by leveraging these probabilistic predicates.

CHAPTER V

DETERMINISM FOR SPECULATIVE APPLICATION DEVELOPMENT

5.1 Introduction

While, STMs has significantly eased the process of writing speculative code they are not a panacea for all bugs. It is still possible to make mistakes while using the transactional construct. The very nature of speculation with aborts, rollbacks and retries can make the debugging process more difficult. For these reasons its is crucial to provide developers with the tools necessary to make writing speculation code easy. STMs have helped developers side-step many of the issues with parallel code, but one of the key remaining challenges which restrict programmer productivity is the non-determinism of parallel code. Non-determinism describes the phenomenon where code behaves differently during different executions despite there being no non-determinism in the inputs. Non-determinism complicates application development.

Serial code is inherently deterministic in nature and is the programming model that most programmers are accustomed to. Deterministic behavior is also much easier to reason about. In parallel applications, however, non-determinism arises due to the parallel application itself and/or due to the parallel machine and its runtime. The difficulties that non-determinism introduces has long been recognized [116]. STMs further amplify these issues by being speculative in nature.

There has been growing interest in applying determinism to parallel applications with several systems discussing a wide variety of techniques [93, 13, 81, 86]. Prior work, has however primarily focused on introducing determinism into parallel applications which use traditional synchronization primitives such as locks. With one of

the major goals of STMs being increased programmer productivity, they serve as a prime candidate for the inclusion of determinism thereby further strengthening their appeal.

While there is a growing consensus that determinism is important, there is still disagreement as to what extent of determinism is desired (and worth paying for). There are in fact a wide range of types of determinism ranging from applications which are just data-race free to those which use synchronous parallelism (parallelism proceeding in lock step). A middle ground which balances between these extremes, that is often settled on, is known as *Internal Determinism* [17] and is what we adopt as well. *Internal Determinism* requires key aspects of intermediate steps of the application to be deterministic.

Another type of determinism is *External Determinism*. Applications which exhibit this type of determinism are only required to produce the same output when run on the same input (execution may vary internally, even at key points, as long as the output is the same) and are said to be *determinate*.

Internal Determinism, the type we adopt, has many advantages. In addition to leading to external determinism [95], it also provides many benefits such as ease of development, ease of reasoning about code, ease of debugging and ease of testing [17]. Internal Determinism requires that key intermediate steps during execution be deterministic. In an STM execution these are transactional commits and aborts and hence we provide determinism at this granularity.

We were able to classify the STAMP benchmark suite [21] into *Externally Deterministic* and *Externally Non-deterministic* applications, as shown in Table 2. In *bayes* the learn score changes between runs. In *kmeans* the clustering varies between runs. In *labyrinth* the paths generated varies and in *yada* the resulting triangulation varies between runs. Note that each of the varying results is correct and acceptable.

Irrespective of whether STM applications exhibit *external* determinism, all STM

Table 2: Classification of STAMP Benchmarks

Externally Non-deterministic	Externally Deterministic
bayes	genome
kmeans	intruder
labyrinth	ssca2
yada	vacation

applications inevitably exhibit *internal* non-determinism. Similar to the non-deterministic order of critical section execution in parallel applications which use traditional synchronization primitives (such as locks), the order of executing transactions is highly non-deterministic. It is these differences in ordering that lead to non-deterministic behavior ¹.

The STM notion of *aborts* introduces yet another source of non-determinism into an already crowded landscape. Aborts ensure correct STM operation by rolling back offending transactions when atomicity constraints have been violated. Hence, while the execution of an instruction in a critical section using locks guarantees visibility of its effects (in some cases at the end of the critical section), no such guarantee can be made for an instruction in an STM transaction due to the notion of aborts. In fact, its behavior in subsequent attempts may be completely different due to intervening commits. Its effects will be made visible only after a successful commit (after all aborts), introducing complications during implementation and debugging [133, 51]. Introducing determinism eases the burden on the programmer significantly.

Let’s take a look at how non-determinism manifests itself in a real world application. Consider the *yada* benchmark from the STAMP STM suite. The *yada* benchmark implements Ruppert’s algorithm for Delaunay mesh refinement [105], a technique for generating unstructured meshes of triangles which satisfy certain guarantees such as bounds on angles. It is not *externally deterministic* since it does not produce the same output every time it is run (however, each output is correct and

¹If the order in which the operations execute changes the result then this leads to *external non-determinism* in addition to *internal non-determinism*

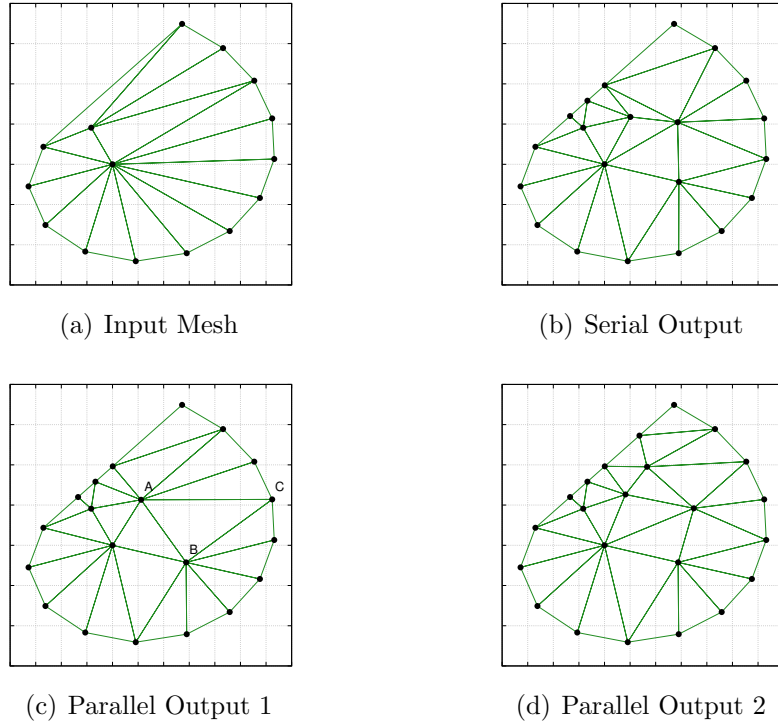


Figure 29: Input and Output Meshes for the *yada* Benchmark

acceptable). It is often used in interpolation, rendering, terrain databases, geographic information systems and also to find solutions for partial differential equations by the finite element method. Figure 29 shows the input mesh (Figure 29(a)), the output mesh during serial execution (Figure 29(b)) and two sample output meshes during parallel executions (Figure 29(c) and Figure 29(d)). Non-determinism complicates application development phases:

Implementation Programmers typically follow an iterative process where they write a piece of code, make sure it works correctly and then loop back continuing the implementation process. This approach implicitly assumes determinism in intermediate execution stages. While this is a given when writing serial code it does not hold for parallel code. Figure 29(b) shows the serial output. This serial output as well as the internal steps used to obtain it never change, irrespective of how many

times it is run ². Non-determinism throws a wrench in the iterative implementation process.

Debugging Consider two sample output meshes obtained during parallel execution (Figure 29(c) and Figure 29(d)). Consider the triangle marked ABC in Figure 29(c). If the application encounters a bug while generating triangle ABC, a typical debugging cycle includes running the application a few times and gathering relevant information about that particular triangle. With non-deterministic execution subsequent executions might not even contain that triangle (as can be seen in Figure 29(d) and even Figure 29(b))! This severely effects debugging.

Testing Testing is often performed by comparing output with a *golden* (correct) output. As one can see from the output meshes in Figure 29 there is no correct output that can be used for comparison in this situation. This hampers testing of code.

Introducing determinism into STMs helps programmers. In this work we first adapt techniques which introduce determinism in applications which use traditional synchronization to work in conjunction with certain STMs (see Section 5.3.1 for applicability). As one would expect, introducing determinism does lead to performance degradation over a non-deterministic execution. Next we present, DeSTM, which uses novel techniques exploiting the properties of these STMs to dramatically improve the performance of deterministic executions. Determinism typically imposes a fixed execution schedule. However, DeSTM also allows programmers to randomly change the deterministic schedule in a controlled fashion giving programmers access to a wide variety of execution schedules during development. We use TL2 [36], a popular STM framework to demonstrate our approach.

We evaluate our approach using the STAMP benchmark suite. We first study the overheads that determinism introduces in STM applications and then demonstrate

²Certain algorithms like random algorithms exhibit varying behavior on each run. However, these are also typically still controlled by a seed or other external factors which should be quantified as *inputs*.

how DeSTM is able to improve performance of deterministic execution significantly, by over 50% in some applications and on average by about 35%. DeSTM also actually helped us detect, what we believe is a bug, in the *vacation* benchmark (this bug pattern helped us identify a similar issue in another benchmark where it manifests itself at lower compiler optimization levels). Further, our approach is programmer friendly and does not require any changes to application code.

The remainder of this chapter is organized as follows. In Section 5.2 we discuss the classical double barrier approach to determinism. In Section 5.3 we discuss some of the general ideas behind determinism in STMs. In Section 5.4 we discuss how to adapt the classical double barrier technique to STMs. In Section 5.5 we present DeSTM. In Section 5.6 we present a comprehensive evaluation (as well as a discussion on the issue in the *vacation* benchmark). We present related work in Section 5.7 and summarize in Section 5.8.

5.2 *Determinism in Multi-Threaded Code*

Double Barrier Technique The classical approach to introducing determinism into a multi-threaded application which uses traditional synchronization primitives (such as locks) involves using a turn based algorithm coupled with double barriers [81]. This gives each thread a turn in which it can make modifications to shared state as shown in Figure 30. Turns are typically implemented using a token passing algorithm. Ownership of the token indicates that the thread may make modifications after which it passes the token to the next thread. The execution is divided into two phases: the *Parallel Phase* and the *Serial Phase* with each Parallel Phase being called a *round*.

Parallel Phase In this phase changes made by individual threads are contained or isolated in some manner (one technique is to convert threads into isolated processes [81]). This ensures that any modifications by threads to shared state are not visible

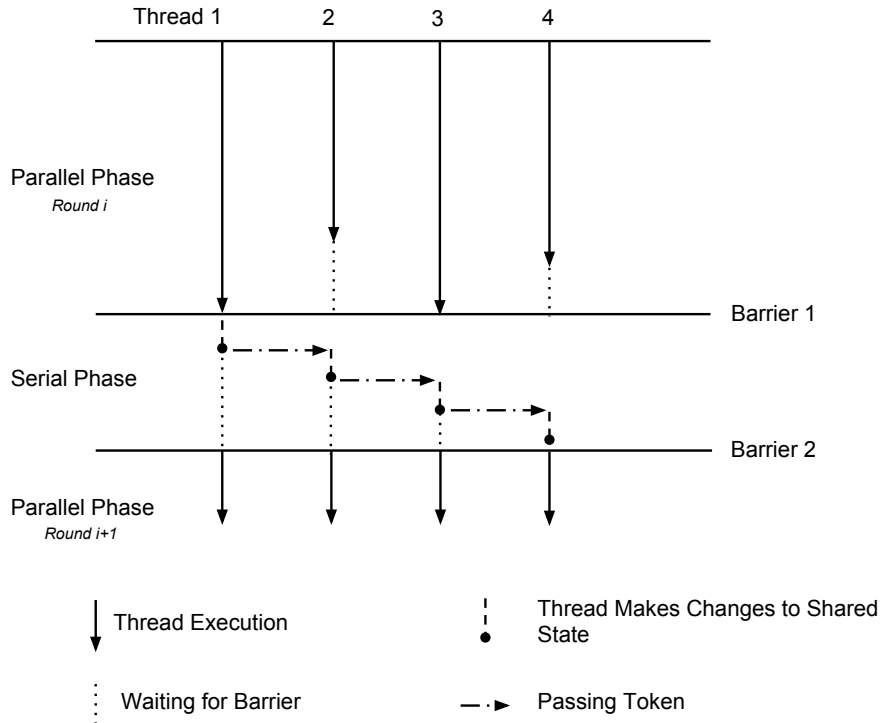


Figure 30: The Double Barrier Technique

to other threads in the parallel phase. Note, threads execute freely without any synchronization in this phase.

Threads continue running until they reach Barrier 1 (the first barrier). The placement of Barrier 1 varies between implementations (systems such as Dthreads [81] use synchronization boundaries such as barriers and locks). Once threads reach Barrier 1 they switch over to the Serial Phase.

Serial Phase In this phase the token passing algorithm gives each thread a turn to make its changes to shared state after which it passes the token to the next thread in a round robin fashion and proceeds to wait at Barrier 2 (the second barrier). Turns ensures that all changes to shared state are made in a deterministic order and the barriers ensure that changes do not effect execution of the Parallel Phase. Once all threads reach Barrier 2 they then begin the next *round* of the Parallel Phase.

Section 5.4 discusses how to adapt this to STMs.

5.3 *Determinism in STMs*

5.3.1 *Applicability and Limitations*

There are a wide variety of STM designs and implementations. In this work we focus on introducing determinism into a popular subset of STMs which use global clocks to provide opacity and commit time locking (Section 4.3 in [54] provides an excellent description of this class of STMs). Commit time locking systems use lazy versioning and make changes to shared state only during commit. Lazy versioning systems buffer updates until commit time and are also referred to as deferred update or write-back systems. A number of STMs employ this design including TL2, DeuceSTM (CTL configuration) [73], SMV [96] and TinySTM (CTL configuration) [121]. Our approach can not be applied to other types of STMs which do not use global clocks such as InvalSTM [52] (uses invalidation). Further, our approach to determinism is applicable to STM applications which use well-formed code (that read/write to shared state within transactions). We built our approaches on top of TL2, a popular STM framework.

5.3.2 *STM Recap*

Before moving on we quickly recap how this class of STM infrastructures work (for a more detailed description see Section 4.3 in [54]). At the start of a transaction, the STM samples the *global clock* and stores it in a per-transaction variable (typically called the *read version*). Within a transaction, reads are logged and occur from shared memory itself (if not previously written to by the same transaction) and writes to shared memory are logged (in a buffer). After the execution of the transaction, the STM attempts to commit any changes, with its *Commit Protocol*. The commit protocol checks if any other threads have modified any locations read to or written by this transaction (by comparing the current version number of the memory locations with the *read version*). If the locations have been modified, it is a *conflict* and the

STM aborts the transaction and retries. Such a situation will arise if any of these memory locations have been modified after the transaction has started. If all the memory locations are unmodified, the transaction locks all of them and commits by writing the changes from its buffer into shared memory, updating the memory locations' version numbers and by incrementing the *global clock*. After a successful commit of a transaction the thread continues its execution past the atomic construct.

5.3.3 Internal Determinism

As discussed in Section 5.1 *internal determinism* requires key aspects of intermediate steps of the application to be deterministic. For STM applications the key steps are transactional commits and aborts. We provide determinism at this level.

The *trace* An execution of an STM application generates a certain commit/abort order. A commit can be represented by the string $T_i\text{thread_number}_i\text{-C}$ and an abort can be represented by the string $T_i\text{thread_number}_i\text{-A}$. A sorted sequence of these strings by time is referred to as the *trace*. For example, $T_1\text{-C:T}_2\text{-C:T}_1\text{-C:T}_2\text{-A}$ describes 2 threads with 3 commits and 1 abort between them. For STM applications which use our deterministic framework this trace is unique. The programmer can also choose to output this trace to aid development.

5.3.4 Invariants

During STM application development, the number of commits and aborts along with other statistics are typically output at the end of an execution. These numbers vary significantly from run to run in a typical execution. However, by introducing determinism into the execution, by virtue of having a unique *trace*, the number of commits and aborts that occur *must* be constant across different runs, hence becoming invariant. This extra piece of information serves as an additional tool to help programmers identify potential issues since varying counts in a deterministic execution point to a bug.

In fact, once we introduced determinism into applications from the STAMP suite we observed that for the *vacation* benchmark, the commit and abort counts varied across runs. This quickly lead us to find, what we believe is a bug, in the actual benchmark code (see Section 5.6.5 for a discussion).

5.3.5 Non-determinism due to Memory Conflicts

There can be unusual sources of non-determinism in STMs which need to be removed. For example, consider the mechanism used to detect conflicts. While different implementations use different mechanisms to implement conflict detection, TL2 maintains versioned write locks for memory locations which indicate whether a location is locked as well as a version number to detect conflicts (as discussed in Section 5.3.2). Memory locations are mapped to a limited number of versioned write locks through a hash function, ensuring that the number of locks needed is modest. Detecting conflicts using this technique can sometimes lead to false conflicts. That is, if the addresses of A and B map to the same versioned write lock (through the hash function), two concurrent transactions one of which operates on A and the other on B will conflict despite not actually conflicting with each other, a false conflict. A false conflict, however, also introduces non-determinism. Let's see why.

While the hash function itself is deterministic the memory locations that serve as input to the hash function are not. Address Space Layout Randomization (ASLR) is a security technique which randomly arranges the positions of key data areas in the application (including the stack and heap) in the process's virtual address space on each run. This causes the virtual memory addresses used in each run to be different.

Figure 31 depicts the scenario where concurrent accesses to two integers A and B results in a false conflict in one case and does not result in a conflict in another. This causes one of the transactions in Run 1 to abort while these particular accesses will not cause either transaction in Run 2 to abort. This leads to different traces and

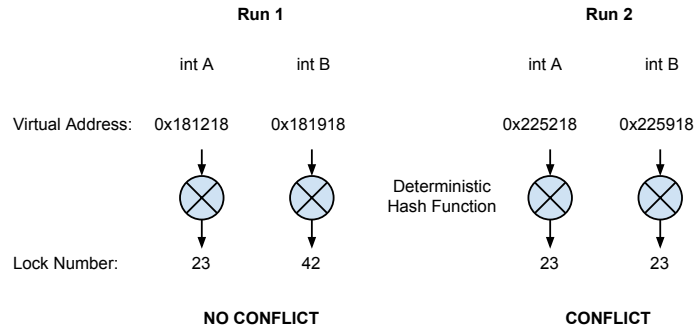


Figure 31: The Effects of ASLR on STM Conflict Detection

non-deterministic execution.

In our prototype implementation we solved this problem by turning ASLR off³⁴. Indeed, ASLR improves security by protecting against certain buffer overflow attacks which our approach would be vulnerable to. A permanent solution to this problem would involve developing an ASLR aware hash for STMs (potentially using differences from the start addresses of memory segments) and is outside the scope of this work.

Depending on the underlying STM implementation other sources of non-determinism such as non-deterministic wait loops, non-deterministic counter and version number updates also need to be disabled.

5.4 Double Barrier Technique for STMs

We now discuss how to adapt the classical double barrier technique (Section 5.2) to these STM systems. To use the double barrier technique we first need to provide isolation to different threads while they are running in the Parallel Phase. While previous techniques have often developed specialized mechanisms (for example by converting threads into isolated processes [81]), the STMs we consider (Section 5.3.1), naturally provide complete isolation through their atomic constructs. All modifications made

³Several techniques can be used to turn off ASLR. For example via the ADDR_NO_RANDOMIZE personality flag or via the *setarch* linux command

⁴GDB also turns off ASLR to aid debugging

to shared data are automatically isolated from other running transactions, while still providing parallelism making them especially suitable for deterministic executions.

The start of the commit protocol of a transaction provides a convenient location to place Barrier 1, switch over to the Serial Phase and begin token passing. This forces a thread to wait for the token in order to begin its commit protocol. Once it owns the token it executes its commit protocol completely and then passes the token to the next thread. The next thread then gets a chance to execute its commit protocol. This causes threads to take turns modifying the shared state in a deterministic fashion. Recall that modifications to shared state are made only during the commit protocol. A successful commit entails locking all modified memory locations, writing the changes from the local buffer, updating their version numbers and incrementing the *global clock*. Note, that at the end of the commit protocol a transaction does not necessarily commit its changes to memory but can also abort if it finds changes to memory it has used which violate its atomicity constraints. However, notice that whether a commit or an abort occurs is in fact deterministic. This is because a thread begins its commit protocol only when it is the token owner, implying that threads before it have completed their respective commit protocols resulting in a deterministic memory state. Barrier 2 is placed at the end of the commit protocol. This ensures that threads can not continue executing before all transactions finish updating their changes to shared state.

Figure 32 depicts this technique. This approach leads to *both* an internally and externally deterministic execution. As one would expect this does slow down execution time when compared to a non-deterministic execution (see Section 5.6 for an evaluation). In TL2, we implemented these barriers using pthread barriers.

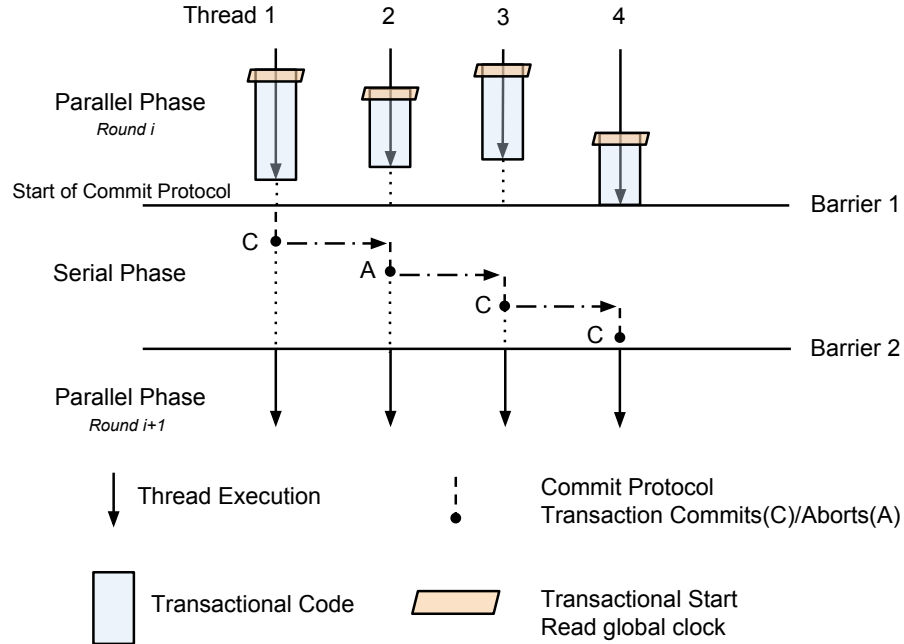


Figure 32: The Double Barrier Technique

5.5 DeSTM

We now present DeSTM, which improves performance of deterministic execution significantly by exploiting properties of these STMs.

Notice in Figure 32 that Thread 1 has to wait for all threads to arrive before it can begin its commit protocol. This requirement was put in place to ensure that changes to shared state are not made while other threads are executing in the Parallel Phase. However, by using this STM model (Section 5.3.2) it is in fact safe to commit changes while other threads are still executing. Let's see why.

Consider the situation in Figure 32 where Thread 1 writes to memory location A in its transaction. These writes are logged by the STM into a buffer. Let us also assume that Thread 2 is writing to the same memory location A in its transaction. Under the double barrier technique, changes made by Thread 1 will be committed to shared memory during its commit protocol. When Thread 2 executes its commit protocol it will observe these changes (by observing the version number on the location's lock)

and abort.

However, these STMs also in fact perform the same operation (checking version number on a location's lock) at every memory read and write during execution of the transaction itself (in addition to the checks in the commit protocol). This allows it to detect atomicity violations early and abort instead of waiting until the commit protocol to detect the violation and abort. Hence, if we allow Thread 1 to commit changes before all threads reach Barrier 1 it is actually safe. At most this will only cause Thread 2 to abort its transaction early in the Parallel Phase itself instead of during its commit protocol in the Serial Phase. Both possibilities still always cause an abort leading to an *internally* and *externally* deterministic execution.

Note, to maintain determinism we still need to pass a token between different threads to maintain ordering. This allows us to relax the requirement of having threads wait at Barrier 1 to begin their commit protocol but rather allow them to begin as soon as they are token owners.

This might suggest that we can get rid of both the barriers entirely and allow a thread to commit when it has the token. However, this is not the case. Let us examine this situation in more detail and see what happens when we try to remove each of the barriers.

5.5.1 Removing Barrier 1

Hypothetically removing Barrier 1, let us assume that the only constraint to start the commit protocol is the ownership of the token. Consider two Threads (1 and 2) running in the Parallel Phase (both in the same round, *Round i*), the first one executing transaction *X* and the second executing transaction *Y*. Thread 1 owns the token which it will pass it to the next thread, Thread 2, when it completes its commit protocol. Figure 33 depicts two possible outcomes in this scenario.

Let's say one of the variables that both these transactions read and write is, say

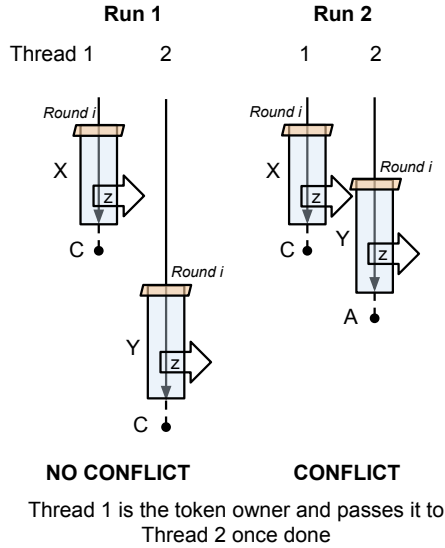


Figure 33: Effect of Removing Barrier 1

z . In *Run 1* Thread 1 finds that it owns the token and hence commences its commit protocol on transaction X and commits successfully. After it commits Thread 2 begins the execution of its transaction Y . In this case, despite the fact that it is reading and writing to the same variable z , since transaction Y starts (reads the current *global clock*) after transaction X commits there is no conflict (since it is reading from the latest value) allowing Thread 2 to commit transaction Y successfully.

Consider an alternative ordering, in *Run 2*. In this situation, Thread 2 begins the execution of its transaction Y *prior* to Thread 1's commit of transaction X . Now, since z is being modified in between the execution of transaction Y , transaction Y will abort (either early during a read/write or during the commit protocol). In effect, this lack of ordering constraints introduces non-deterministic behavior. To solve this problem we can add a constraint:

Constraint 1 *A transaction may begin its commit protocol when it owns the token and only after every other transaction has started in that round (i.e. after each has read the current global clock).*

Constraint 1 solves the race condition of whether other transactions have started

in a round, leading to a deterministic execution while still being significantly relaxed over having a complete barrier, Barrier 1.

In our implementation built on TL2, Constraint 1 is implemented by using a synchronized start counter on a per round basis. After a transaction successfully starts it increments this counter. When a transaction wants to commit it first ensures that all threads have started by waiting until the count is equal to the number of running threads (taking into consideration new threads starting and threads ending).

5.5.2 Removing Barrier 2

Barrier 2 makes every thread wait after the execution of its commit protocol for every other thread to complete its commit protocol. Considering that the STM is protecting reads and writes, would removing Barrier 2 continue to provide determinism? After all threads would still need to wait for the token to begin their commit protocol for the next round. However, removing Barrier 2 entirely does not lead to a deterministic execution. Let's see why.

Hypothetically removing Barrier 2, consider two threads running in the Parallel Phase, with Thread 1 executing Transaction X in *Round i* and Thread 2 executing Transaction Y in *Round $i - 1$* . This can happen after Thread 1 has completed its execution of the previous round and has moved on to the current round while Thread 2 is still working on a transaction in the previous round. Note, that in this situation Thread 2 is still the token owner as it has not yet completed execution of Transaction Y in *Round $i - 1$* . Figure 34 depicts two possible outcomes in this scenario.

Similar to the previous example, let's say one of the variables that both these transactions read and write is z . Thread 2 finishes the execution of Transaction Y in *Round $i - 1$* and passes the token to Thread 1. In *Run 1*, Thread 1 happens to start (reads the current *global clock*) its transaction after Thread 2 has committed its transaction in the previous round. In this situation there is no conflict. However,

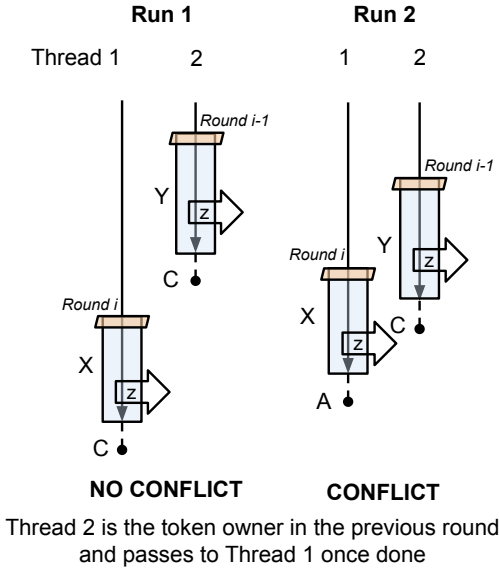


Figure 34: Effect of Removing Barrier 2

in *Run 2*, Thread 1 happens to start the transaction *prior* to Thread 2 committing transaction *Y*. Hence, when Thread 1 attempts to commit transaction *X* it will see that the value of *z* has been modified since it started leading to an abort (either early during a read/write or during the commit protocol). The possibility of these two outcomes leads to non-determinism. To solve this problem we can add a constraint:

Constraint 2 *A transaction may start **only** after every other transaction in the previous round has completed its commit protocol.*

Constraint 2 solves the race condition between transactions in the previous round and the current round, leading to a deterministic execution while still being significantly relaxed over having a complete barrier, Barrier 2.

In our implementation built on TL2, Constraint 2 is implemented by using a synchronized commit protocol counter on a per round basis which threads wait on to start a transaction. Once set by the last thread in the previous round (taking into consideration new threads starting and threads ending), threads pass through this check.

Constraint 1 and 2 coupled with the token passing algorithm are sufficient to guarantee a deterministic execution. Constraint 1 and Constraint 2 replace Barrier 1 and Barrier 2. These constraints are very relaxed when compared to the requirements that full barriers impose. Barrier 1 requires all threads to complete execution of a transaction before any of them can start the commit protocol, whereas Constraint 1 is considerably more flexible. Barrier 2 requires each thread to wait at the end of its commit protocol for all other threads to complete their commit protocols before it can continue. Constraint 2, is again, considerably more flexible than this.

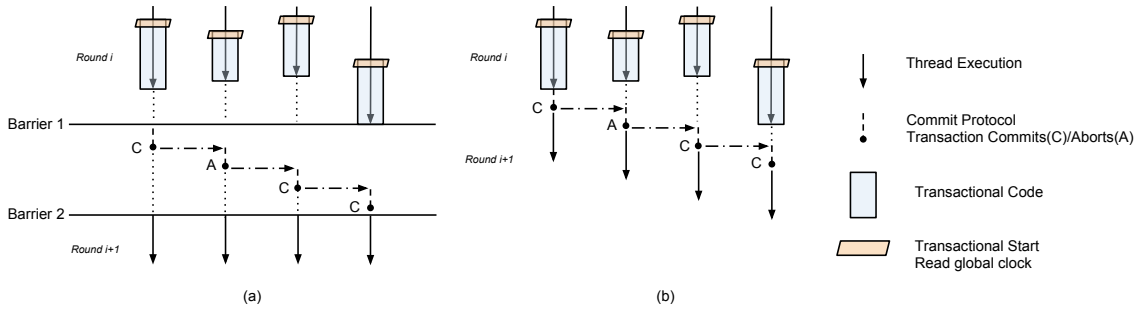


Figure 35: Comparison Between the Two Techniques

Figure 35 compares the two execution models. Figure 35(a) shows the execution of 4 threads executing transactions using the double barrier technique adapted to STMs and Figure 35(b) shows the same execution under DeSTM. We can clearly see that removing the two barriers and replacing them with more flexible constraints leads to significantly improved execution time in *Round i*.

5.5.3 Randomizing the Execution Schedule

Token passing follows a round robin approach, being passed from Thread i to Thread $(i+1)$ (wrapping around). Figure 36 presents the pseudocode. The *next_token_owner* array contains the next thread that each thread should pass the token to (For example, *next_token_owner*[2] = 3 implies Thread 2 will pass the token to Thread 3). Note, for this discussion thread indices are 0 based.

```

// Stores the current token owner.
int token_owner;

// Stores next thread each thread passes token to.
// For example: next_token_owner[2] = 3;
// Implies that thread 2 will pass the token to thread 3.
int next_token_owner[NUM.THREADS];

// Initialize with a round robin pattern.
initialize() {
    for (i = 0; i < NUM.THREADS; ++i) {
        next_token_owner[i] = (i + 1) % NUM.THREADS;
    }
}

// Look up the next_token_owner array and pass the token.
pass_token() {
    lock token_owner;
    token_owner = next_token_owner[thread.id];
    unlock token_owner;
}

```

Figure 36: Token Passing Mechanism

Under such a scheme the token always passes in a round robin fashion (Figure 37(a)). Each round uses the same pattern. This leads to a single deterministic execution schedule.

To expose other deterministic schedules to the programmer, DeSTM provides a flag which can be used to randomize the *next_token_owner* array for every round. However, a simple approach of just randomizing the array does not work. Consider the randomized *next_token_owner* array shown in Figure 37(b). Despite the array being randomized the passing pattern produced by this randomization has *multiple cycles*, between Thread 0 and 3 as well between Thread 1 and 2.

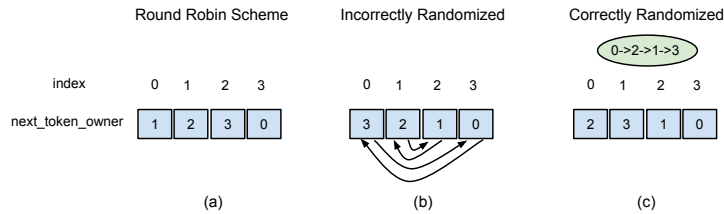


Figure 37: Randomizing the *next_token_owner* Array

To generate a correctly randomized *next_token_owner* array DeSTM first generates a random *Passing String* (shown in the green oval in Figure 37(c)). This can be generated by a simple random permutation of a string containing all thread identifiers. The *next_token_owner* is then generated from this string. This ensures that the

next_token_owner array is correct, containing only a single cycle with every identifier. Generating a new *Passing String* every round allows DeSTM to generate a new execution schedule each round ⁵.

The randomization is controlled by a single random *SEED*. The programmer can modify the random order provided by varying the *SEED*. This randomization technique allows the runtime to expose numerous execution schedules for use during implementation, debugging and testing.

5.6 *Evaluation*

In this section we present a comprehensive evaluation of the double barrier approach and DeSTM. Our prototype implementation of DeSTM is built on top of TL2 (0.9.6). We use the STAMP benchmark suite (0.9.10).

All experiments were performed on a dual socket machine with two Intel Xeon CPU E5540 processors at 2.53GHz with 4 cores each for a total of 8 cores. The machine was running Ubuntu 12.04.3 LTS (Kernel Version: 2.6.35-32-generic). All experiments were compiled with gcc 4.6.3 with the O3 optimization level. All experiments are the average over 10 runs. All benchmarks were run with recommended inputs ⁶.

5.6.1 **Determinism**

We first experimentally verified that the execution traces (see Section 5.3.3) were unique across a large number of runs for all STAMP benchmarks. We ran 200 runs of each STAMP benchmark using the large recommended input and 20000 runs of each benchmark using the small recommended input with both the Double Barrier approach and DeSTM. All runs executed deterministically producing the same output

⁵Due to a limitation in our current implementation every passing string we generate must begin with the first thread identifier (Thread 0), however the rest of the passing string is random.

⁶As specified in benchmark README files

and trace (external *and* internal determinism) ⁷.

5.6.2 Performance

Next we compare the performance of these techniques:

- **Non-Deterministic** This the baseline non-deterministic implementation consisting of the STAMP benchmark suite running on top of TL2 with no modifications. It does not link with any of our libraries.
- **Double Barrier (DB)** In this approach we adapt the classical double barrier technique to work with TL2 as described in Section 5.4.
- **DeSTM** This version provides deterministic executions using the techniques described in Section 5.5.

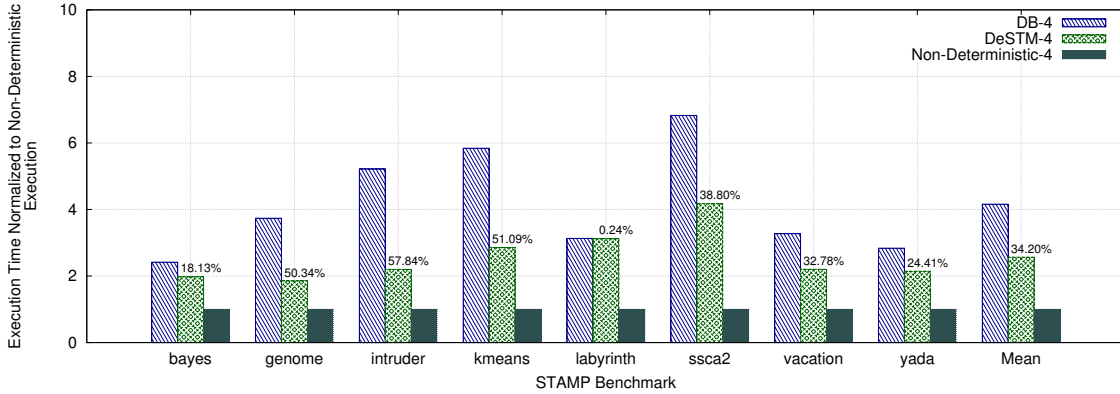


Figure 38: Performance Comparison Between DB Approach, DeSTM and Non-Deterministic Executions

Figure 38 reports the results we obtained. The results report the execution time of *DeSTM* and the *DB* approach normalized to the execution time of the non-deterministic execution. Each benchmark was run with 4 threads (degree of parallelism) for each technique (including the Non-Deterministic execution). The labels

⁷One benchmark, *vacation*, had what we believe is a bug, which first needed to be fixed. See Section 5.6.5 for details.

on the DeSTM bars indicate the percentage performance improvement of DeSTM over the DB approach.

Discussion We see that when determinism is applied to STMs by adapting the classical Double Barrier (DB) approach overheads are relatively large (sometimes more than 6x). On an average the execution time of the DB approach is little more than 4x that of the non-deterministic execution with the same number of threads.

DeSTM is able to significantly improve on execution time of the classical DB approach by more than 50% for some benchmarks (*genome*, *intruder* and *kmeans* benchmarks) and on an average by almost 35%. On average the execution time of DeSTM is only a little more than 2 times that of the non-deterministic execution.

The performance gains that DeSTM exhibits varies between benchmarks. Recall, that DeSTM relaxes the requirements of the two barriers replacing them by simpler constraints which leads to less time idling and more parallelism, translating in to performance benefits. We see that in all the benchmarks DeSTM outperforms the DB approach. The *labyrinth* benchmark exhibits only a small performance gain under DeSTM (0.24%). This is due to its extremely unbalanced workload in which some transactions are extremely short while some are extremely large. This high imbalance overpowers any benefits that can be gained by DeSTM. However, in all the other benchmarks DeSTM exhibits significant improvements over the DB approach.

Also, note that in no situation does DeSTM perform worse than DB. The constraints that DeSTM imposes are always more relaxed than having barriers hence always leading to better performance.

Overall, the overheads exhibited by deterministic executions using DeSTM are modest. With non-determinism being one of the key challenges to parallel application development we argue that these overheads are certainly acceptable during application development.

5.6.3 Scalability

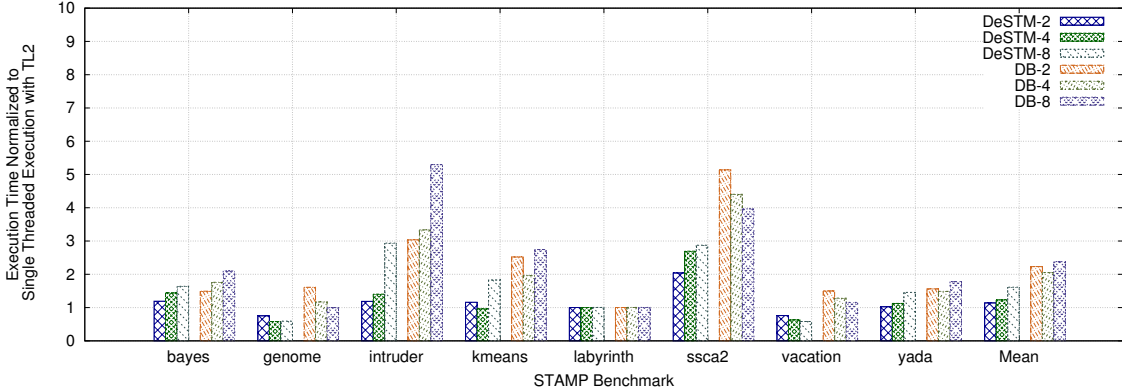


Figure 39: Scalability of the DB and DeSTM Executions

In this section we report results for execution time with a varying number of threads. Figure 39 reports the results we obtained. The execution times are normalized to the execution time of a single threaded execution with TL2 (which does not link with any of the code which induces determinism).

Discussion Increasing the number of threads in a deterministic execution typically causes more synchronization overheads. This is because, there are now a larger number of threads to wait for either during barrier synchronization (DB approach) or while waiting for the constraints to get satisfied (DeSTM). Imbalances in execution time between transactions hence get magnified leading to higher synchronization overheads. On the other hand, increasing the number of threads also serves to decrease the amount of work each thread needs to do. The results in Figure 39 represent the effective outcome of this trade-off and is highly application dependent. We observe that some benchmarks demonstrate improved performance with increasing number of threads (*genome*, *vacation* and up until 4 threads *kmeans*) while others do not. The *labyrinth* benchmark shows constant execution time regardless of the number of threads. This is because in addition to exhibiting highly unbalanced behavior (as explained in Section 5.6.2), the benchmark exhibits very high contention patterns,

allowing only one or two transactions to commit per round regardless of number of threads used.

The scalability trends between the DB approach and DeSTM are similar. A very interesting observation is that *even the slowest DeSTM execution* on a given benchmark is *faster than the fastest execution using the DB approach* demonstrating that DeSTM is able to reduce overheads in deterministic executions quite significantly.

5.6.4 Overheads

While the two previous sections compared the performance of deterministic executions to non-deterministic executions, in this section we provide a breakdown of the overheads in a deterministic execution. The experiments were run with 4 threads. The results we obtained are reported in Figure 40.

Discussion As discussed in Section 5.5, DeSTM improves the performance of deterministic execution by removing the requirements of the barriers and replacing them with more flexible constraints. Figure 40 shows the different components of the overhead in deterministic executions and illustrates how DeSTM is able to effectively reduce time waiting at the barriers. Notice how the percentage of overhead due to waiting due to the barriers is much larger than that of the waiting due to the constraints. This significant reduction in time spent at barriers allows DeSTM to effectively reduce execution time of deterministic executions.

Note again, how the highly unbalanced transaction profile in *labyrinth* causes almost all the overhead in the DB approach to occur while waiting for Barrier 1 (the other two components are present but are not visible due to scale).

We also measured the overheads due to randomizing the execution schedule (see Section 5.5.3) but it was less than 1% in each of the benchmarks and hence we omitted these overheads in this discussion.

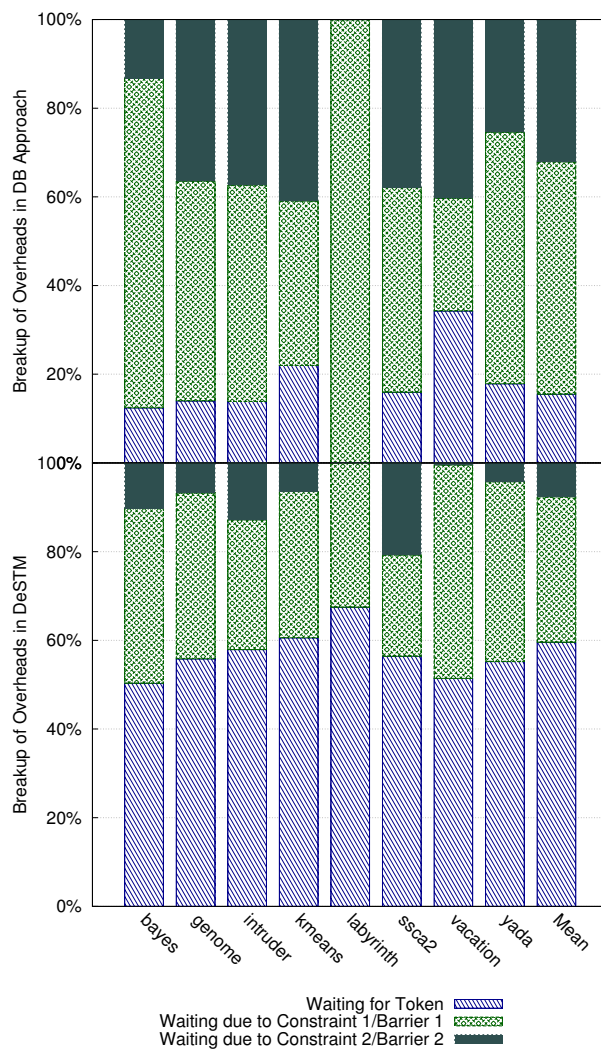


Figure 40: Breakdown of Overheads in Deterministic Executions

5.6.5 Invariant Based Bug Identification

As discussed in Section 5.3.3 the introduction of determinism into an STM application causes every execution to have a unique trace, by virtue of which the number of commits and aborts also remain constant in every run. During STM application development, these numbers are typically output at the end of an execution. While, these values change significantly between runs in a typical execution, with determinism they do not and this fact can potentially serve as a valuable debugging tool. When these counts (or in general the execution trace) vary between deterministic runs, it

points to the existence of a bug in the code. Bug free code *will* have a unique trace. The differences between the execution traces serve as an extremely helpful starting point to locate and fix the bug.

While we were running the STAMP benchmarks with DeSTM, we observed that for the *vacation* benchmark [21] these counts (and also the execution traces) varied between runs, suggesting a bug. Further investigation quickly lead us to find, what we believe is a bug, in the benchmark code.

```

1 void client_run(void* argPtr) {
2     ...
3     // Part of a larger switch statement inside a loop.
4     // The following two arrays are declared as locals on the stack.
5     long maxPrices[NUM_RESERVATION_TYPE] = { -1, -1, -1 };
6     long maxIds[NUM_RESERVATION_TYPE] = { -1, -1, -1 };
7     ...
8     TMBEGIN();
9     ...
10    // This is inside a loop.
11    if (price > maxPrices[t]) {
12        maxPrices[t] = price;
13        maxIds[t] = id;
14        isFound = TRUE;
15    }
16    ...
17    if (isFound) {
18        Add Customer; // Adds customer information.
19    }
20    Perform operations with Customer;
21    ...
22    TMEND();
23    ...
24 }

```

Figure 41: Original *vacation* Code

```

1 void client_run(void* argPtr) {
2     ...
3     ...
4     TMBEGIN();
5     // Initialize these inside.
6     long maxPrices[NUM_RESERVATION_TYPE] = { -1, -1, -1 };
7     long maxIds[NUM_RESERVATION_TYPE] = { -1, -1, -1 };
8     ...
9     TMEND();
10 }

```

Figure 42: Fixed *vacation* Code

Figure 41 shows the original code. The *maxPrices* and *maxIds* arrays (local variables) are used to find the maximum prices of items and their corresponding identifiers. A typical *if* check is used to identify the maximum price on *line11*. STMs typically provide a *LOCAL_WRITE* API which should be used when writing to local variables so that these writes are rolled back correctly when a transaction

aborts. However, in this case the API calls were inadvertently missed causing the values in the array to restore incorrectly (subsequent retries after an abort use old values)⁸. This in turn caused the addition of the Customer on *line18* to sometimes not occur (causing insertions for certain customers to be missed). This in turn led to different traces with different commit and abort patterns.

Once the variability in the traces was identified (DeSTM outputs lots of debugging information in traces which includes transaction instances which caused variance) we were able to pin point the original transaction which caused the variant behavior and the initialization issue which caused it. While there are several approaches to fix the issue, we resorted to simply moving the initialization code for the two arrays *maxPrices* and *maxIds* inside the transactional scope. This ensures that when the transaction restarts after an abort the arrays are reset correctly. Figure 42 shows the changes made to the code.

Based on this bug pattern we looked for potential issues in other benchmarks. A similar issue is also present in the *bayes* benchmark at the O0 optimization level, however unlike the *vacation* benchmark, at higher optimization levels the offending variable is optimized away by the compiler, hiding the bug. Nonetheless, it is not safe to depend on the compiler to optimize away the variable. *genome* also exhibits a similar issue but unlike *bayes* or *vacation* it does not effect correctness.

⁸Sometimes regardless of whether the the *LOCAL_WRITE* API is used, the values of local variables will restore correctly. Such a situation arises when the local variable in question is stored in a register. STMs implement abort/retries with the *longjmp/setjmp* functions (or its variants) which restore registers correctly after a rollback (*longjmp*). Whether a local variable is stored in a register or not is compiler dependent and hence this behavior cannot be relied upon to restore values. Portable methods include declaring the variables inside the transactional scope or using the *LOCAL_WRITE* API while writing to the variable. In the above example code, *isFound* is stored in a register and hence gets restored correctly while the two arrays *maxPrices* and *maxIds* do not, leading to an inconsistent state after a rollback.

5.7 *Related Work*

There has been a burgeoning interest in determinism for parallel applications. While determinism and the challenges it imposes has a rich history dating back to at least the 1960s [72, 48, 95, 116] there has been a surge of interest recently in approaches which seek to remove non-determinism from parallel applications from several angles including hardware [33, 60, 34], operating systems [11], programming languages/models [19, 17] and several runtime approaches [93, 132, 13, 14, 81, 82]. With the growing importance of parallel programming in today's multi-core architectures the need for leveraging determinism to ease development is clear.

Olszewski et al. presented Kendo [93] one of the first systems to provide determinism for race-free programs by serializing synchronizations in a deterministic order. Similar to their approach our system provides deterministic guarantees until the first race in the code. One of the strengths of such techniques is that it allows programmers to systematically eliminate all races in an iterative process [93]. CoreDet [13] and Dthreads [81] present techniques to introduce determinism into arbitrary multi-threaded code which rely on the POSIX threading APIs and discuss the classical double barrier technique. Others have [86, 82] built on this work and discuss how to improve performance by relaxing synchronization requirements. However, all of these works have aimed to provide determinism to parallel applications which use traditional synchronization techniques such as locks. As such they are not applicable to code which uses software transactional memory due to differences in the nature of abstractions and due to the predominantly low-level synchronization primitives used in STM implementations. STMs have been found to be an extremely useful tool in reducing the complexity of parallel programming. Integrating determinism in STMs is crucial in further easing the development of parallel programs and is a theme we explore in this work.

Bocchino et al. [19] present DPJ (Deterministic Parallel Java) and argue that

parallel programming should be deterministic by default and present a type and effect system for Java. Blelloch et al. [17] present a programming model for “nested-parallel” programs through commutative and linearizable operations. StreamIt [119] is able to provide determinism by virtue of the streaming model that it adopts. Cilk [45] is able to provide determinism for a certain subset of legal programs and its Nondeterminator race detector tool [25] can be used to identify data races. DeSTM on the other hand introduces determinism into applications that are being developed using the transactional programming model.

With transactional memory increasing in popularity there has been increased interest in aiding the development process. Gottschlich et al. discuss techniques used to debug transactional applications [51] and present a visualization tool in [50]. Debuggers [133] and novel performance bottleneck analysis tools [134] for transactional memories help programmers during the development cycle. While, non-determinism has been long recognized as one of the key challenges to parallel application development, its interplay with STMs has been relatively unexplored. Determinism in STMs was discussed in the preliminary work of Smiljkovic et al. [114] which proposes using the classical double barrier approach (similar to our DB approach but with extra barriers at transaction start). On the other hand, in this work we present DeSTM which exploits properties of certain STMs, removes strict synchronization requirements imposed by barriers and improves performance of deterministic executions significantly while also allowing programmers to specify multiple deterministic execution schedules.

Record and Replay (RnR) [78, 88, 90, 129] techniques can be used to replay previously recorded executions, another important tool used to assist the debugging process. RnR techniques for STMs have also been developed [51].

5.8 *Summary*

In this chapter we discuss how non-determinism complicates the development of speculative applications and how introducing determinism can significantly ease the burden on the programmer. We discussed why it is especially powerful to couple it with STMs which strive to improve parallel programmer productivity. We presented DeSTM, an infrastructure which allows programmers to leverage determinism through the implementation, debugging and testing phases of STM application development. We discussed the classical double barrier approach and presented DeSTM which further exploits the properties of certain STMs to prevent undue performance loss. Non-determinism is one of the remaining key challenges that parallel programmers face during development, and we believe that DeSTM enhances the ease of developing speculative applications using STMs.

CHAPTER VI

EFFICIENTLY SUPPORTING DISTRIBUTED ALGORITHMIC SPECULATION

6.1 Introduction

As it becomes increasingly difficult to extract more parallelism from applications through traditional means at the application level, speculation brings a refreshingly orthogonal view to the problem. Different proposals to exploit speculation at the application level exist. Prabhu et al. introduced a programming model which can speculatively parallelize loops by using value speculation [97]. Software Transactional Memory (STM) systems [54] and models such as Galois [75] allow programmers to speculate about dependencies and the concurrency of different operations.

High-level (also known as algorithmic or function-level) speculation is another type of speculation that can be introduced by the programmer. The algorithmic speculative paradigm allows programmers to exploit deeper algorithmic properties to extract parallelism. Many important applications demonstrate variance in execution time even on the *same input*. Other applications are amenable to being solved through multiple approaches with varying execution times. For these kinds of applications, programmers can use high-level speculation to speculatively run multiple instances of the (same or different) function or algorithm *in parallel* and then dynamically choose to incorporate the results from the best one. We will refer to this kind of speculation as *high-level speculation* and it is the focus of this chapter.

Speculation leads to an easier path from sequential code to parallel code. Designing parallel algorithms is tough, writing a parallel program (based on parallel algorithms) is tougher, writing a correct parallel program is even tougher and writing

a correct and highly efficient parallel program is the toughest! But per Amdahl's law overall performance will not scale unless the least parallelizable parts can scale up. So the question is: how do we support efficient speculation that scales with the number of available processors.

High-level speculation has shown to improve performance significantly for a wide variety of applications [28, 122, 99]. Programming models and frameworks which support high-level speculation aim to provide the requisite infrastructure to allow programmers to quickly and efficiently exploit high level speculative parallelism with minimal effort. However, the current programming models and frameworks are severely limited by both the amount of speculation that they support and their ease of use. The amount of speculation that these models are designed to support is strictly limited by the amount of parallelism available on a single machine. Their designs do not allow for scaling outside a single machine. For example, if the machine being used has 8-cores, the maximum number of parallel speculative threads that can be run at a time is 8. Over-provisioning threads to cores serializes speculations defeating their purpose. To speculate at large scales moving to a distributed (cluster) environment is inevitable. Distributed speculation, with the fundamental shift from shared memory to distributed memory, requires a completely different approach and new designs. Distributed speculation brings along with it a host of transparency, efficiency, scalability and ease of use issues. In this chapter we present several novel techniques and elegant mechanisms to tackle these non-trivial issues. Like previous work [28, 122, 99] our goal is to provide a framework which programmers can easily leverage to incorporate high level speculation into their applications with minimal effort. We now briefly discuss and motivate the need for large scale distributed high-level speculation while empirically demonstrating the inadequacy of current models.

6.1.1 Motivation

While variance in application execution time based on input data is expected, many exhibit variance even on the same input data. There are a wide variety of reasons why a variance may arise. For example, it may be due to randomness in the algorithm. It may be due to the ability to solve a problem using multiple approaches, where each approach has a drastically different execution time. Each of these multiple approaches may even produce results with different levels of desirability (quality). Non-deterministic behavior may even be due to issues such as scheduling. This variance in execution time is the key to speculatively parallelizing these applications. To leverage this kind of variance at the application level the programmer can *branch the execution of a program at a speculation point into multiple independent speculations running in parallel*. From these parallel speculations, *the best one is chosen and its results are used as the execution continues past this speculative region* (depicted by Figure 44). Let's now see why previous approaches which are limited to speculation (using the parallelism) in a single machine are not sufficient to extract all the available parallelism in the application.

Let us consider an application which needs to solve a SAT problem during its execution and let's say it uses the WalkSAT [111] algorithm to solve it. WalkSAT uses random assignments and flips to find satisfying solutions. Due to the random nature of the algorithm, WalkSAT exhibits dramatic variability in execution time. Figure 43(a) plots the execution time of 500 runs of WalkSAT on a sample DIMACS [106] dataset ¹. A majority of the runs exhibit high execution times while a few of them exhibit dramatically reduced execution times. This is simply due to the random nature of the algorithm. Note again that these execution times are for the same input dataset and also note the log scale on the y-axis. Figure 43(b) plots the cumulative

¹Dataset used was par16-5.cnf

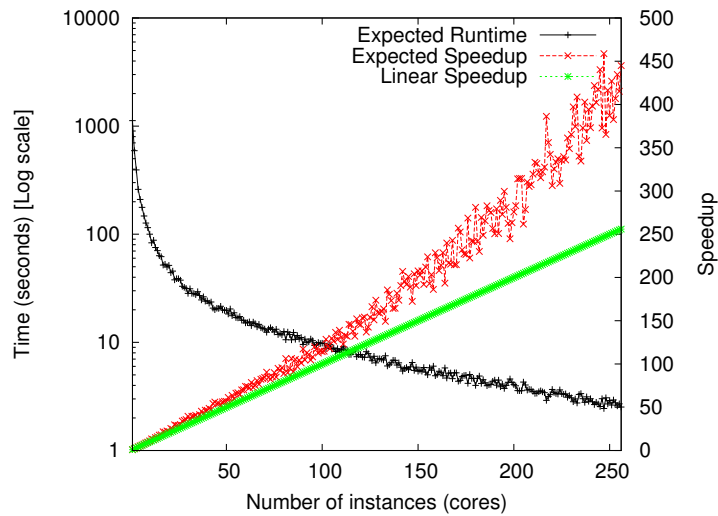
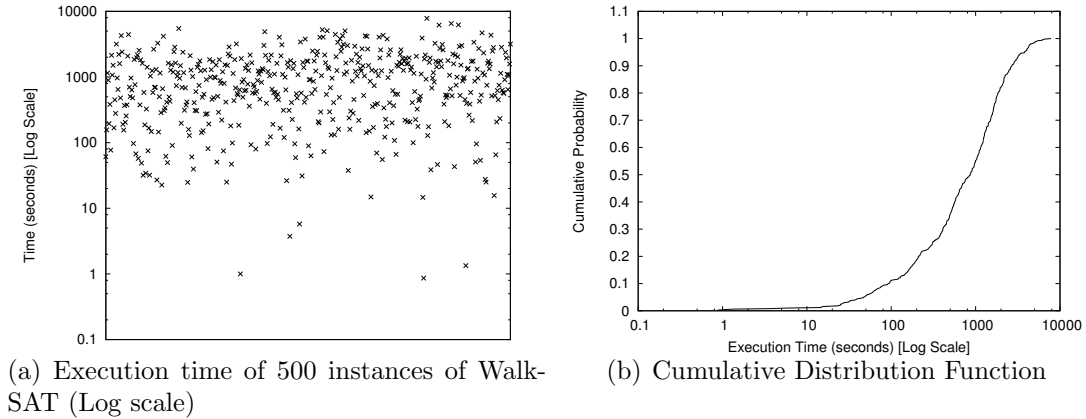


Figure 43: WalkSAT Execution Time, CDF and Speedup

distribution frequency (CDF) of the execution time. It shows the probability (y-axis) that the runtime will be less than a particular value (x-axis). We observe that the probability of getting a very low execution time is fairly small.

Instead of running just a single instance of WalkSAT to solve a SAT problem, if the original application were to speculatively run multiple instances of the WalkSAT function in parallel and choose the fastest (best) one it can get significant performance improvements. To quantify the improvements of execution time of the WalkSAT function in our example we need to compute the expected execution time with respect to the number of parallel instances. We assume that each individual instance of the function runs on a single processing core. To get the expected execution time for

X cores we take X random values from the 500 execution samples and choose the minimum value as the expected execution time (averaged over a large number of tries). Figure 43(c) shows the expected execution time with an increasing number of instances (up to 256). Significant speedup can be observed as the number of instances increase. In fact, the speedup is super linear! This is simply due to the wide variability in the underlying execution times and the CDF for this example. Importantly, we observe *good scalability going well past the typical number of cores on a single machine*. This points to the fact that *speculation outside a single machine has significant benefits and can enable sustained scalability at large scales*.

In this chapter we propose the Multiverse speculative programming model for the C language. Multiverse is designed from the ground up to deal with large scale distributed speculation. We present abstractions and a runtime which allow programmers to introduce large scale speculation into applications with minimal effort. The programming model allows for multiple independent speculations to run concurrently in a single address space (across a cluster) while transparently maintaining isolation between them. Multiverse provides single commit semantics, ensuring that the changes made by only the best speculation transparently becomes visible to the rest of the program as it continues its execution past a speculative region. Multiverse allows programmers to scalably harness the available parallelism across an entire cluster and is not limited by the parallelism in a single machine.

6.1.2 Multiverse Execution Model and Terminology

Let's consider an application that can benefit from the speculative execution model (Figure 44). In general it can be represented by the code prior to the speculation (*pre-speculation code*), the speculative code itself (each individual instance is called a *speculation*) and the code that executes after the speculation (*post-speculation code*). The *speculative region* is the part of the code where multiple options are speculated

on (and run in parallel). *Speculative boundaries* separate speculative regions from the rest of the code. If n speculations are launched in a speculative region, then the *degree of speculation* is said to be n . Only changes made by the best speculation are visible to post-speculation code while others are discarded.

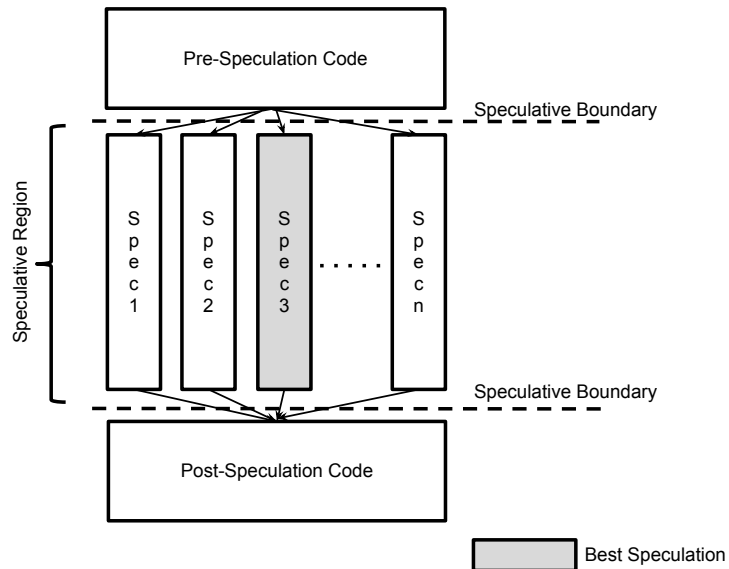


Figure 44: Basic Multiverse Speculative Execution Model

Each individual speculation can execute the same code (as in the WalkSAT example) or it could execute different pieces of code. For example, if an application wanted to perform sorting speculatively, each individual speculation could run a different sorting algorithm. An application may have multiple speculative regions. Multiverse enables speculative execution even across machine boundaries (without the presence of physical shared memory) while still providing a single unified address space for all speculative and non-speculative code to run in. This means that speculative code does not need to be written in any special way and still gets transparent access to the original address space of the program. It even supports the use of arbitrary pointers across speculation boundaries. This is a very powerful abstraction and is key to the Multiverse framework as it allows programmers to write speculative code just like normal code with minimal effort.

Multiverse is designed to harness the parallelism across a typical cluster comprising of a set of compute nodes (the word *node* is used interchangeably with *machine*) each with some number of processing cores. The nodes are connected by a high speed interconnect without the presence of any physically shared memory.

The remainder of this chapter is organized as follows. In Section 6.2 we go over the usage of the Multiverse programming model. In Section 6.3 and Section 6.4 we discuss Multiverse’s design and how it achieves its goals. In Section 6.5 we discuss scalability issues. Section 6.6 evaluates Multiverse over several benchmarks and discusses other performance implications. We present related work in Section 6.7 and summarize in Section 6.8.

6.2 Multiverse Usage

Multiverse allows programmers to easily describe speculative regions in code. To understand what Multiverse allows programmers to do and how it does it, let us consider an application which solves a SAT problem using WalkSAT. Let’s say the original application invokes the WalkSAT function with some arguments like:

```
walksat(param1, param2,  
        problem_ptr,  
        solution_ptr);
```

Let’s say that this function exhibits variance in execution time which can be exploited through speculative parallelization. Multiverse allows the programmer to speculate on multiple instances of it by simply wrapping it with the keywords *speculate_with* and by specifying the degree of speculation (in this case 100).

```
speculate_with(100, walksat(param1, param2,  
                            problem_ptr,  
                            solution_ptr));
```

This can be read as *speculate with a 100 instances of this walksat invocation*. Let's look at this function call in a slightly larger context and see how exactly this would work and the challenges it involves. Consider the code fragment in Figure 45 (while this is just demonstrative pseudo-code it contains all necessary modifications to use Multiverse).

```
#include "multiverse.h"

solve_sat(SatProblem *problem_ptr) {
    // Some stack variables.
    int param1, param2;

    // Allocated on heap.
    SatSolution *solution_ptr = malloc(sizeof(Solution));

    // Speculate with 100 instances of WalkSAT.
    speculate_with(100, walksat(param1, param2, problem_ptr, solution_ptr));

    // A non-speculative execution would look like:
    // walksat(param1, param2, problem_ptr, solution_ptr);

    // Verify the solution.
    verify_sat(problem, solution);
}

// Code which exhibits variance.
walksat(int param1, int param2, SatProblem *problem_ptr, Solution *solution_ptr) {
    ... Read/write arbitrary data.

    ... Read and solve problem from problem_ptr.
    loop {
        ... Read/write arbitrary data.
    }

    solution_ptr->solved = 1;
    solution_ptr->solution = <update solution>;
}
```

Figure 45: Example Multiverse Usage

When execution reaches the *speculate_with* call, Multiverse takes charge, creates 100 different instances (speculations) of the *walksat* function and launches them on multiple machines as needed. Like any normal function, each individual *walksat* function has access to variables in its scope (this is non-trivial to achieve for processes on different machines). It uses *param1* and *param2* to control its behavior. It reads the SAT problem from *problem_ptr*, solves it and updates the solution pointed to by *solution_ptr* (which was dynamically allocated on the heap in *solve_sat*). Each

speculative invocation of the function can independently modify variables and only changes made by the best speculation will be made visible to post-speculation code.

Due to the cluster environment, the original application and the speculations are all implemented as distinct processes. The process running the original application is referred to as *Process0* and the others which execute speculative code are referred to as *speculative processes*. MPI (the OpenMPI implementation) is used for communication.

Ease of use: In this example, the addition of the *speculate_with* (and corresponding `#include`) is the only code change that needs to be made. *speculate_with* is syntactic sugar and is implemented as a macro which makes calls to the Multiverse runtime. Note that Multiverse *does not require programmers to annotate or mark or wrap variables for cross machine/speculative access* and requires no complicated changes to the code. This makes Multiverse fairly easy to use for any programmer.

Implications: This simplicity has several implications on the runtime. Speculative processes need to continue execution from exactly the point that Process0 began the speculation. Data modifications need to be private to each speculation. Changes made by *only* the best speculation need to be made visible.

Isolation between speculations is easily satisfied by the use of distinct processes with private address spaces. The biggest challenge is now providing the illusion of a single address space to each of these speculations in different processes across machines scalably. We discuss how Multiverse achieves this in Section 6.3 and discuss scalability in detail in Section 6.5.

6.2.1 Limitations

While Multiverse tries to make it as easy as possible to incorporate high level speculation into applications, it does have some limitations:

- Code executing in speculative regions must not perform I/O.
- Code must use *malloc/free* (and not *mmap* or custom allocators unless tuned for Multiverse).
- More generally, code in speculative regions must not use system calls with side effects as Multiverse does not support transparent access of open sockets, files and other system specific constructs across speculative boundaries.

Further, our current implementation has the following limitations:

- Speculative code must not use global variables (we discuss workarounds in Section 6.3.4).
- Multiverse does not support nested speculation.
- No support for multi-threaded code across speculative boundaries.
- Our implementation requires a 64-bit address space.

6.3 *Multiverse Design*

Multiverse enables transparent execution of speculations in a distributed setting by using a unique on-demand address space sharing mechanism. To understand how this works let us look at what describes the execution state of a process at a particular point in time. It is the content of its address space as well as its execution context. The address space consists of the code, global, stack and heap sections (as shown in Figure 47(a)), note that we are referring to the virtual address space of a process here. The execution context of a process contains the contents of machine registers amongst

other items (for example it contains the current stack pointer and program counter). Speculative processes need a copy of the address space and execution context at the speculative boundary from Process0. Multiverse sends copies of these to speculative processes when a *speculate_with* call (a speculative boundary) on Process0 is reached. Once the best speculation completes, Multiverse performs the reverse operation by copying that speculation's address space and execution context back to Process0 to continue execution.

During execution of the speculative region Process0 acts a *Speculative Region Server*. The Speculative Region Server is the Multiverse component in charge of orchestrating the execution of different speculations (including starting speculations, stopping speculations and determining which is the best speculation). This execution model is depicted in Figure 46.

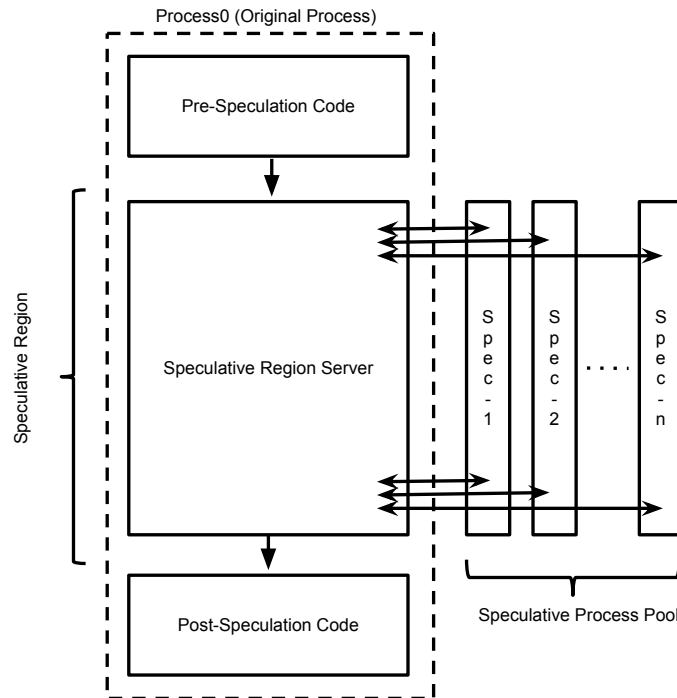


Figure 46: Multiverse Speculative Execution Model

This is the basic idea behind Multiverse's execution model. However, as one

can imagine this approach comes with many difficulties and can potentially be very inefficient at a large scale. We discuss how Multiverse makes this solution practical and efficient.

Large Unused Virtual Address Space In principle, a 64-bit processor can address 16 exabytes of memory with a 64 bit virtual address. Most operating systems and applications will not need such a large address space for the foreseeable future so implementing such wide virtual addresses simply increases the complexity and cost of address translation with no real benefit. Current implementations limit the virtual address space to only the least significant 48 bits of a virtual address and require that the remaining bits be sign extended [92], resulting in 256 terabytes of virtual address space. Recall that each process running on a machine has its own independent virtual address space fully available to it. Most applications don't even use a tiny fraction of this address space. Multiverse leverages this fact and uses this large unused virtual address space to enable transparent execution of speculations.

6.3.1 Code Section

Since Multiverse does not place restrictions on functions that are called in speculative regions, the code section in speculative processes must be identical to that of Process0's. There are two broad approaches to get the same code section into another process at the start of a speculative region. The first approach is to launch multiple identical processes with the same code section as Process0, all at Process0's startup time. In this approach all processes except Process0 go to sleep and wait to be woken up (at the start of a speculative region). The second approach is to spawn new speculative processes when a speculative region is encountered in Process0. While conceptually the second approach seems cleaner, spawning many new processes across a cluster every time a new speculative region is encountered is inefficient and incurs too much overhead. The first approach of keeping prelaunched

processes in a sleep state is more efficient since the spawning is done only once and in practice these sleeping processes take up virtually no resources. Hence Multiverse uses this approach. This is akin to the use of thread pools instead of individually spawning threads as and when they are needed [47].

Therefore when a Multiverse enabled application is launched, multiple copies of the same executable are started up on multiple machines to form a pool of available processes that can execute speculative regions of code. These processes are referred to as the *Speculative Process Pool* (SPP). Note that all the processes in the SPP do not need to participate in a particular speculative region but participation is based on the degree of speculation as specified by the programmer. Those processes of the SPP which participate in a speculative region are referred to as *Participating Speculative Processes* (PSPs). *mpirun* [94] is used to start up processes in the Speculative Process Pool. *mpirun* is a highly configurable OpenMPI command that can be used to effectively control the distribution of the pool across the cluster and is highly optimized to launch MPI processes onto thousands of machines.

Now that we have a mechanism to get the code section on to processes in the SPP we can shift our focus on how to move the execution context and stack.

6.3.2 Execution Stack and Context

The execution context describes the current execution state of an application. It contains the process's machine registers, signal mask and a pointer to the current execution stack. The execution context can be used to allow processes in the SPP to continue exactly at the point that Process0 left off. The current prototype uses several Linux system calls to manipulate the execution context: *setcontext*, *getcontext*, *makecontext* and *swapcontext* [62]. Multiverse uses these calls to get a copy of the execution context at speculative boundaries and to move them between processes as needed.

To enable transparent execution of PSPs a copy of the execution stack from Process0 is also needed. Multiverse allows programmers to use pointers to stack locations transparently across speculative boundaries. This means that pointers must point to valid locations as they move from Process0 to the PSPs and back. To do this there are two different approaches.

In the first approach, Multiverse can create a copy of the execution stack from Process0, reload it at a free memory location in the speculative process and then translate all the pointers to point to the right memory locations. Scanning the stack and translating pointers is inefficient and can be inaccurate as it may not be possible to accurately discover and translate pointers in all situations. Hence we use a second approach, in which the stack from Process0 is copied into *exactly* the same address in the PSPs. In this approach, there is no need to translate any pointers since all the memory locations are in exactly the same locations as in Process0 and is hence more efficient. Note that since we are always referring to the *virtual address space* of a process and not the *physical address space* we are guaranteed of its availability even on a different machine.

To provide the illusion of transparent execution between speculative boundaries two distinct stacks need to be maintained for each process. One on which application code executes, the *multiverse stack*, and one which runtime code can use when application code is not executing, the *standard stack*. Multiverse allocates the multiverse stack in an unallocated section of memory. During startup Multiverse ensures that this section is free across all process and reserves necessary memory using the *mmap* [62] system call (a 64-bit virtual address space makes this easily achievable). At this point there are two stacks mapped into the address space on all processes, as shown in Figure 47(b). Multiverse switches between them as needed and only one stack will be in use at a given time. We refer to this behavior as “stack switching”. When processes startup (Process0 as well as processes in the SPP) all of them begin

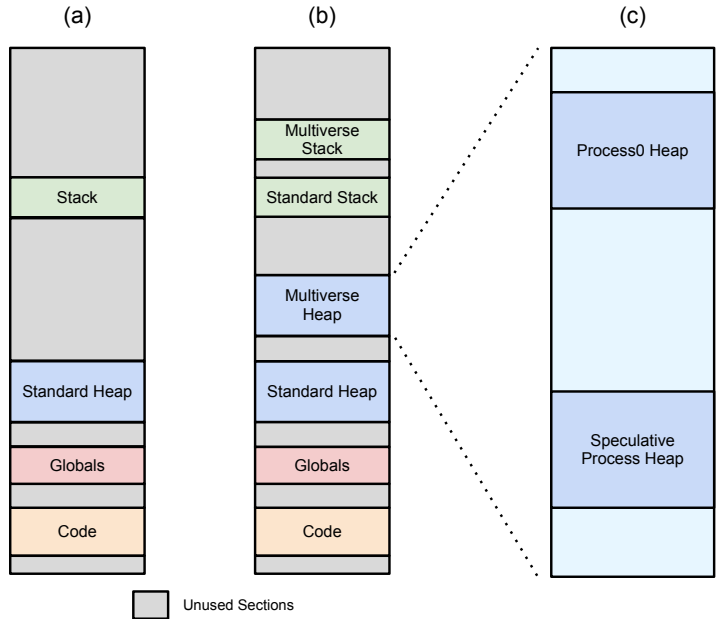


Figure 47: Virtual Address Space

their execution on the standard stack (regular C execution stack). The behavior past this point is different for Process0 and for processes in the SPP. We discuss “stack switching” in the next couple of paragraphs (Figure 48 depicts this for pseudocode).

Process0 Process0 starts up in the standard stack. However, before the first user instruction in `main` even runs, Multiverse transparently creates a fresh multiverse stack and switches the execution context onto this stack. The application code as written by the programmer executes in this *new multiverse stack* (shown in Figure 48(a)). Multiverse performs context and stack switching using the `makecontext` and `swapcontext` system calls [62]. As execution continues and a speculative boundary is reached, Multiverse saves the current execution context and switches the context back on to the standard stack. Subsequent Multiverse runtime code executes on the standard stack. At this point Process0 changes its role to become the Speculative Region Server (still on the standard stack). It selects PSPs from the SPP and sends them a copy of the current multiverse stack and the saved execution context.

Processes in the SPP Processes in the SPP start up in the standard stack as well (Figure 48(b)), and begin their execution by calling into the Multiverse framework. Multiverse puts these processes to sleep and waits for a message from Process0 to begin execution of a speculative region. Once such a message is received the process wakes up and becomes a Participating Speculative Process (PSP). The PSP then waits for a copy of the current multiverse stack and execution context from Process0. Once received, the multiverse stack is directly mapped into memory (at the multiverse stack address) and the saved execution context from Process0 is used to switch execution of the PSP over to the multiverse stack. At this point the PSP is executing on a copy of the multiverse stack from Process0 (with pointers intact). Execution continues until the end of the speculative region at which point a call to Multiverse is automatically made.

The reverse operation is performed at the end of a speculative region after the best speculation is chosen. The context and multiverse stack contents are sent from the best PSP back to the Speculative Region Server which then maps it into Process0's multiverse stack and switches the execution back.

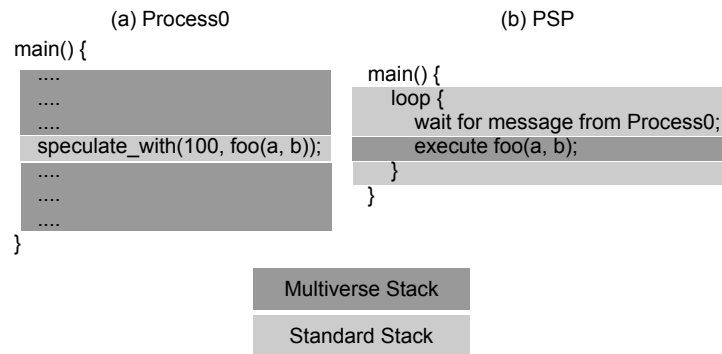


Figure 48: Stack Switching (Multiverse Stack and Standard Stack)

Note that this method only copies the multiverse stack between processes. The standard stack is left in tact in processes and is never copied or copied into. The multiverse stack becomes the execution stack for the application code (as specified by

the programmer) whereas the standard stack serves as the stack that the Multiverse runtime uses.

So far we've dealt with moving the stack to and from Process0 and PSPs and transparently dealing with pointers to stack memory. Typically stack sizes are small and transferring the stack between processes does not incur a large overhead (evaluation in Section 6.6.6). The next step is to deal with dynamically allocated heap memory.

6.3.3 Heap Section

C applications use *malloc/free* to dynamically allocate heap memory. The stack is a limited resource and when a large amount of memory is needed, allocations occur on the heap. This makes transparent access to the heap section a fundamentally different problem from the smaller fixed stack section. With a large and varying heap allocation it might not make sense for Multiverse to transfer all of it over to PSPs because code executing in the speculative region might not even need access to all of it. Predicting before hand the memory that is needed by PSPs is impractical in situations where pointers are used and can vary based on the situation. To combat these issues Multiverse uses a novel on-demand sharing mechanism where only the heap memory from Process0 that is actually needed during execution of a PSP is transparently transferred over.

Multiverse creates special heap sections in the address space, shown in Figure 47(c). All *malloc* allocations happen in these specially allocated sections. These sections are all allocated at page boundaries and are the same across all processes. Since the virtual address space (independent to each process running on a machine) is so large, Multiverse can easily find unused addresses and map it for use. Multiverse provides custom *malloc,free* implementations which are transparently hooked into

the program to ensure allocations happen in these sections. Heap allocations in Multiverse behave differently based on whether they are occurring during the execution of regular code (in Process0) or during execution of a speculative region (in a PSP).

Heap Memory allocated on Process0 Heap allocations on Process0 happen in the *Process0 Heap*. When a PSP is executing code in a speculative region it might need access to some memory from this heap. Multiverse provides transparent, efficient on-demand access to this memory. Before a PSP starts executing, Multiverse protects the entire memory range in the Process0 Heap section in its address space using the *mprotect* system call [62]. The *mprotect* call ensures that any accesses to these memory locations will lead to segmentation faults. The PSP can start executing even without any of the heap memory, it simply needs the execution context and the stack (which have been made available to it). Once the execution on a PSP encounters a heap memory access, a segmentation fault (due to the *mprotect* call being used) will be generated. Multiverse installs signal handlers at the start of execution on processes in the SPP to catch these segmentation faults. If a signal handler catches a segmentation fault it first computes the page that the fault is arising on (based on the address of the fault and the system wide virtual memory page size). It then checks if that is in Process0's Heap section. If so, an MPI request is sent over to Process0 which is currently behaving as the Speculative Region Server (pseudo-code in Figure 49).

Page Server: In addition to acting as the Speculative Region Server, Process0 also takes on the role of a Page Server during the execution of a speculative region. The Page Server services requests for pages in the Process0 Heap. Note that the Page Server is executing in the context of Process0 itself and as such has direct access to all of Process0's memory. If a request for a heap memory page is received from a PSP it simply sends over the entire page of memory that is being requested. The Page

Server does not need to do any heavy book keeping but only simple validations.

Once the page is received by the PSP it maps it into its memory at the same location (obviating the need to update pointers), updates its dirty page information and continues execution. All pages that are received from the Page Server are marked as dirty. When the speculative region completes and the best speculation (best PSP) is chosen, all the pages that have been fetched from Process0 in the best PSP are transferred back to Process0. This ensures that all changes made to these memory pages are available for the post-speculation code. An optimization, which has not been currently implemented, is to mark pages as read only when first brought in and then mark them as dirty only on a write fault. Thus reducing the number of pages that need to be transferred back to only those which have been modified.

```
void signal_handler(void *address) {
    if (!PSP) {
        return;
    }

    if (!(address in valid range)) {
        return;
    }

    void *basePageAddress = computePageAddress(address);

    send to PageServer request for basePageAddress;
    receive from PageServer basePage contents as basePage;

    map basePage into basePageAddress;

    update dirtied_pages information;
}
```

Figure 49: Pseudo-code for Signal Handler

Apart from being transparent to the programmer this approach of transferring memory provides multiple benefits:

- Once a page is brought into memory by the signal handler, subsequent accesses to that page will *not* generate another fault. This implies that only one fault will be generated per page of memory that is accessed by the PSP.
- Since the memory transfer is done at a page level there is much use of spatial

locality. Transfers done at a per-variable/object basis introduce too much traffic and slows down execution of the PSP significantly.

- Different PSPs might need to access disjoint areas of the heap section and this approach ensures that only relevant memory is transferred to each individual PSP.
- Process0 might have large objects/structures allocated in the heap which are not related to the speculative region code at all and hence do not need to be transferred.

Heap Memory allocated in a PSP One practical alternative is to disallow sharing of memory that is allocated in a PSP. Recall that memory can always be allocated in the pre-speculative code and passed in as a pointer and the PSP would have complete transparent access to it.

To allow heap memory that is allocated in a particular PSP to be available even after a speculative region, Multiverse uses a separate Speculative Process Heap (as shown in Figure 47(c)). All malloc allocations in any PSP happen in this section independently. This section is protected using *mprotect* as well. Before a PSP starts execution, all the malloc/free state information from previous speculative region executions are transferred to it (along with the execution context and stack). For new memory allocations an unused page is found in this section, protections are removed on the page, and it is used. If accesses are made to memory that was previously allocated in the Speculative Process Heap the access page faults as in the previous example and the page is brought in from Process0 and execution continues. Note that only the memory allocations made by the best speculation are transferred back to Process0 along with its malloc/free state information.

In this method all heap memory allocated in the best speculation is transferred back to Process0 at the end of a speculative region. An alternative could be to

make the best PSP the new Process0. However, this would either entail transferring the remainder of Process0's address space to the best PSP or it would necessitate each of the processes in the SPP (which have been best PSPs in the past) to act as page servers (answering page requests for their memory) throughout their execution, slowing down future speculative region executions. Multiverse, hence, simply transfers back all allocated heap memory to Process0.

One glaring issue with this entire approach is that Process0 is in charge of fielding page requests from all PSPs and could potentially become a bottleneck. We discuss how to solve this scalability issue in Section 6.5.

6.3.4 Globals Section

As pointed out in Section 6.2.1, Multiverse does not support transparent access of information stored in the global section (including static data) across speculative boundaries. While it is technically possible to simply copy the global section and restore it on the PSP (similar to how the stack section is dealt with) this is generally not desirable. The global section typically contains machine specific information like the communication environment which should not be moved around.

This implies that in our implementation the programmer cannot access globally visible data across speculative boundaries. An easy way around this is to privatize global variables by passing them around in a structure. If this is not feasible to do or the programmer is not willing to, automatic source-to-source conversion techniques which privatize global variables such as [91] can be used to transparently perform this conversion. In any case, the use of global variables is widely considered a bad programming practice and its use is highly discouraged [127].

6.4 Complete Execution Flow and API

We shall now summarize the entire execution flow (shown in Figure 50). At startup time Process0 starts executing application code and processes in the SPP go to sleep

until a message is received from Process0. Process0 executes application code normally until it encounters a speculative region. At the start of the speculative region a copy of the current execution stack and context is made and Process0 switches to the role of Speculative Region Server/Page Server as it orchestrates the execution of the speculative region. The Speculative Region Server sends a start message to PSPs in the SPP with copies of the execution context, stack and malloc information. The PSPs awake from their sleep and switch their execution context to that of Process0's. They execute the code in the speculative region. Any faults to unallocated memory in the PSPs are caught and sent over to the Page Server on Process0. The Page Server handles memory requests from PSPs and sends back the requested memory.

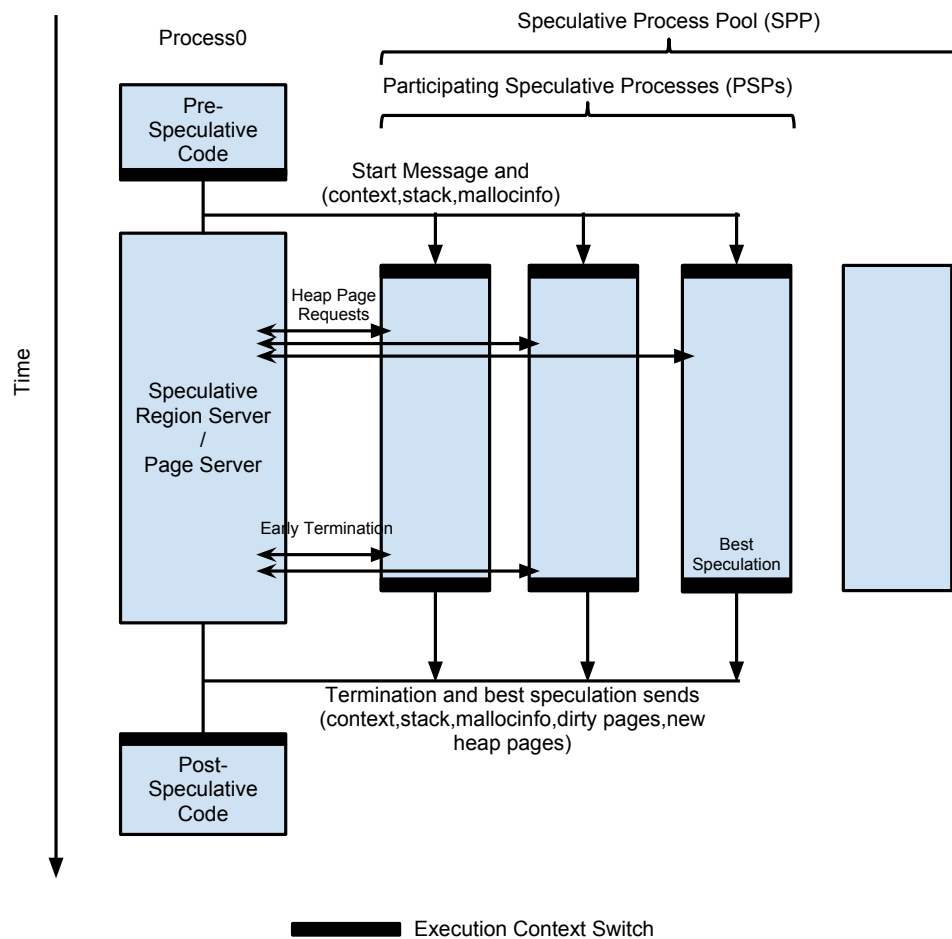


Figure 50: Detailed Execution Model

Execution Termination When a PSP finishes execution, Multiverse will automatically send a notification to the Speculative Region Server. If the best speculation is not to be determined by which speculation finishes first but rather the quality of a result generated, a call to a Multiverse API specifying the quality (desirability) can be made (see Section 6.4.1). The Speculative Region Server waits until it can find a best speculation. Once it does, it will send a message to all other executing PSPs to terminate execution early. All PSPs periodically check for such a message using a timer interrupt (programmer can modify default interval). On an interrupt Multiverse simply checks for a terminate message and if found the PSP's execution is stopped. Multiverse automatically sets up timer interrupts for PSPs.

The best speculation as chosen by the Speculative Region Server, sends over its execution context, stack, dirty pages and newly allocated heap pages. All speculations discard their states after executing and go back to sleep waiting for another speculative region to begin.

6.4.1 API and Usage

Multiverse has a fairly simple API. To make use of the framework the application should include header files and link with the library. A speculation with 100 instances of *foo* can be started with:

```
speculate_with(100, foo(a, b));
```

To start speculations with different functions simply use the following syntax:

```
speculate_with(100, foo1(a, b),  
              100, foo2(a, b, c),  
              100, foo3(d));
```

If these calls are used the best speculation is automatically defined to be the one which finishes first. Multiverse also supports the use of an arbitrary *quality*

parameter. In this case Multiverse will continue executing speculations until at least one of them produces a result with *quality* greater than or equal to the specified value (in the following example *quality* is 42):

```
speculate_with_quality(100, foo1(a, b),  
                       100, foo2(a, b, c),  
                       42);
```

Similar APIs for *quality* less than and equal to are also available. Multiverse uses the end of function execution to automatically determine the end of the speculative region execution on a PSP. However, if we need to return a *quality* parameter or if we need to update the current *quality* metric (during execution) with the runtime, the following APIs can be used:

```
multiverse_running_with_quality(43);  
multiverse_completed_with_quality(43);
```

Other API calls are available which allows code to inspect if it is running in a speculative region and if so which speculation number it is and the degree of speculation.

Nested speculations are not allowed at this point of time. A discussion of possible designs for this are outside the scope of this work.

Alternative Implementations A Checkpoint/Restart design was an alternative we considered. There has been much research in checkpoint/restart schemes over the years [101, 38, 1]. However, checkpointing typically incurs many magnitudes higher overhead than our current approach simply because the entire address space is checkpointed and then reloaded. Spawning processes during the start of every speculative region across an entire cluster and reloading the checkpoint file was simply not scalable in our experiments. Alternative implementations such as the use of an

existing global/distributed shared memory framework for Multiverse internals was too heavy weight and had numerous restrictions which were unsuitable for us. They often required programmers to annotate shared variables, did not allow for access to stack variables across processes and none were designed to deal with scalability issues that arise uniquely in the speculative model (see Section 6.5).

6.5 Scalability

In this section we discuss how we tackle the issue of scalability. There are two potential bottlenecks for scalability. One is in transferring the stack and execution context to all PSPs across a cluster at the start of a speculative region. The other is due to Process0 taking on the responsibility of answering page requests (for heap memory) from all PSPs. The second concern is far more worrisome since large amounts of heap memory can potentially be accessed by numerous PSPs quickly making Process0 a bottleneck. Let us first address this second potential bottleneck.

6.5.1 Heap Memory Accesses

It should be noted that the contents of the address space (heap section) that is being fetched from Process0 is completely static and does not change until the end of the speculative region. Each individual PSP can make its own changes to this memory but those changes will not be visible until the end of the speculative region (after the best speculation is chosen). We can leverage this fact by making copies of all allocated heap pages in multiple processes and machines. This alleviates the need to always go to Process0.

This still does incur some copy overhead, since before the PSPs begin execution we need to duplicate all allocated heap pages into several “copy” processes (across different machines) so that no single machine becomes a bottleneck. However, one more critical optimization can be made. There really is no need to create copies of all pages, especially if most of it won’t be used. Instead of setting up a full copy in

these “copy” processes we can make each of these “copy” processes instead act as a *cache* for page requests. We refer to these processes as *Page Server Caches* and we organize them in the form of a tree, the *Page Server Cache Tree*.

Page Server Cache Tree (PSCT) The organization of the tree is shown in Figure 51. The page server caches are organized as an m -ary tree. PSPs are grouped together in bunches of n and are served by 1 page server cache. Each page server cache serves all page requests from processes in 1 PSP Group as well as page requests from all its children page server caches. Process 0 acts as the root page server cache (Page Server Cache 0).

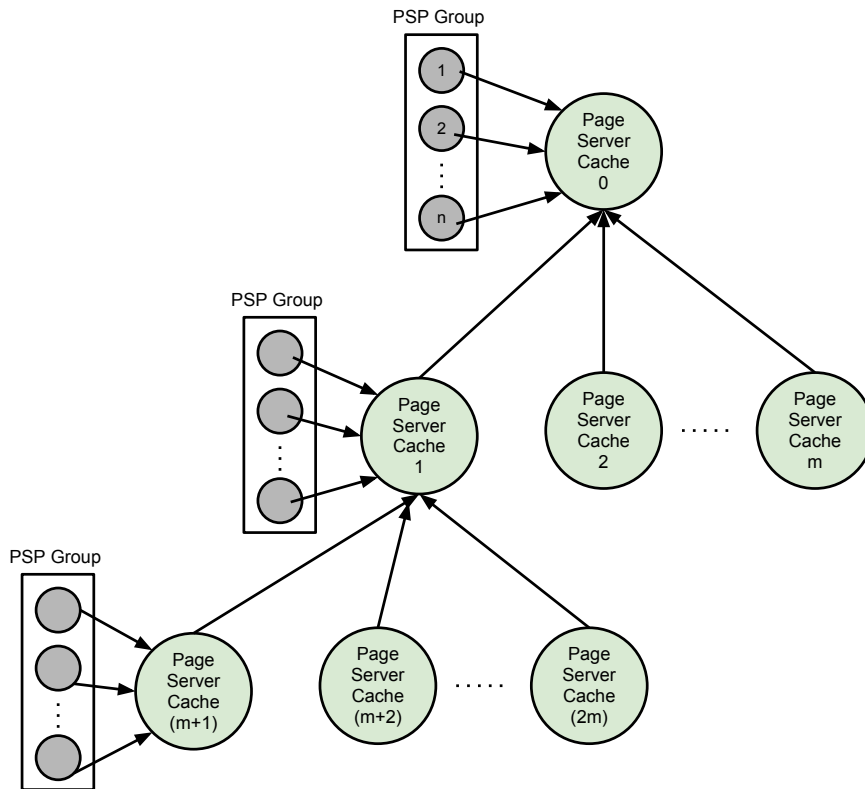


Figure 51: Page Server Cache Tree

Operation The operation of an individual page server cache is in fact similar to the way heap memory is brought in on PSPs. At the start of a speculative region

it *mprotects* the relevant portions of the address space. When a request for a page comes in from a PSP (or any child) it simply tries to access a piece of data from the page. If the page has not been mapped into memory a segmentation fault will occur, the signal handler will be called and the page will be fetched. The page is *not* fetched directly from the root page server cache but from its parent server in the tree. Once the page is fetched it is mapped into memory (just like a PSP) and it sends the page back to the requester. Subsequent requests to the same page will not cause segmentation faults and hence will be serviced directly.

Note that this operation is performed recursively up the tree. For a freshly set up tree on the very first request, segmentation faults will happen all the way up to the root node and as the page is being sent down each server keeps a copy in its address space. This has the nice property of automatically creating copies in multiple nodes so that subsequent requests under those nodes don't go up the tree. This ensures that at no point is the root server overwhelmed (see Section 6.6.6 and Appendix A). In the default configuration each page server cache serves 8 PSPs and the PSCT is organized as a binary tree.

6.5.2 Stack and Execution Context

The stack and execution context needs to be transferred to all PSPs at the start of their execution. This is not as much a concern because the stack size is limited and fixed compared to the arbitrary large sizes the heap can take on. The transfer of the stack and execution context is performed at the start of a speculative region to all PSPs. Our initial approach (Linear approach) of Process0 transferring this information to each PSP in a serial, sequentialized fashion was not scalable to larger cluster sizes. We therefore, switched over to a tree-like algorithm (Tree approach). The processes are arranged in a binary tree rooted at Process0 with each parent node disseminating copies of the stack and execution context to its children. This tree

based approach alleviates any concerns of excessive overheads for these transfers (see Section 6.6.6).

6.6 Evaluation

In this section we present a comprehensive evaluation of the Multiverse framework. We demonstrate the utility of large scale distributed speculation using several benchmarks which demonstrate execution time variance. We also analyze the overheads of the Multiverse framework and its ability to scale.

We integrated the Multiverse framework prototype into several benchmarks: WalkSAT, All Interval Series, Costas Arrays, Perfect Square and TSP (Traveling Salesman Problem). The metric for choosing the best PSP for the first four benchmarks is execution time, while the metric for the TSP benchmark is quality based and is an acceptably low cost path. Modifications to the code were minimal (as shown in Figure 45) and required almost no understanding of the algorithm itself.

Benchmarks have phases such as initialization phase, a speculative phase and verification and output phases. The verification and output phases (present in the original code as well) verify in memory data structures ensuring correct operation and display detailed operational statistics of the algorithm based on many statistics stored in memory over the course of the execution.

While one might choose to use a simple script which launches many copies of the benchmark executable and collects results from the one which finishes first (killing others) instead of using Multiverse, such an approach would not achieve the expected results (for example: due to benchmarks' complete loss in ability to perform any non-revocable operations and operations with side-effects such as input/output) without several code modifications, would not be practical for benchmarks such as TSP (which use a quality metric for completion instead of execution time) and would also introduce redundant code execution (non-speculative phases) across the cluster (leading

to resource wastage at large scales). Multiverse enables large scale distributed speculation through a cleaner, more efficient approach and is generally applicable as a framework to other C codes as well.

Methodology We measure the speedup obtained using Multiverse with a varying number of PSPs. When Multiverse runs it uses a certain numbers of PSPs (each on 1 core) and 1 Process0 (an additional core). Speedup is defined as the ratio of execution time of the original application without using Multiverse to the execution time of the application using Multiverse. The original application without Multiverse ran a single instance of the function which exhibits variance (like a regular execution flow) on a single core and did not link with Multiverse at all. A speedup of 1 indicates that both versions had the same execution time. This is indicated by the dotted line (Baseline 1) in the graphs which show speedup. The speedup graphs show speedup numbers with varying number of PSPs (+1). For example, if the X-axis value shown in the graph is 64, then there are 63 PSPs along with the 1 Process0 each running on individual cores. Each PSP in a given benchmark accessed the same area of the heap (as other PSPs that were launched in parallel). All benchmarks were run on two different clusters:

- **A 86-core 43-node cluster (Cluster A)** Intel Dell PowerEdge 1850 Linux cluster with dual Pentium4 Xeon EMT64 processors (3.20GHz) using Infiniband interconnects. The cluster was running Linux version 2.6.18 and running gcc 4.1.2.
- **A 1176-core 98-node cluster (Cluster B)** With each node containing two 2.6 GHz six-core AMD Opteron processors (Istanbul) using the Cray SeaStar2+ interconnect. The cluster was running the Cray Linux Environment (CLE) 3.1 and was running gcc 4.6.2.

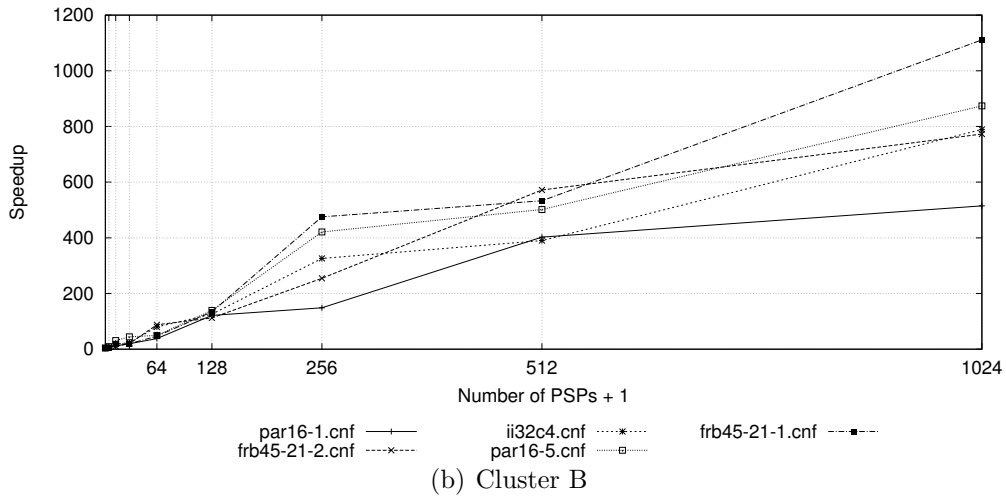
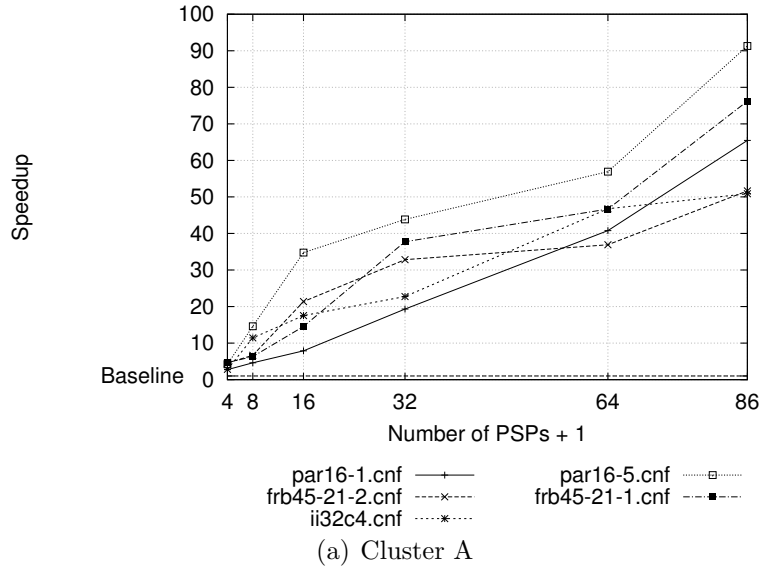


Figure 52: Results from WalkSAT Benchmark on different datasets

We wanted the PSCT processes to share the same cores that PSPs were running on (not to use additional cores). We compared performance between using additional cores and multiplexing PSCT processes onto PSP cores on Cluster A and observed no difference in performance (PSCT processes are small in number and not CPU intensive). However, Cluster B’s Cray ALPS scheduler prevented over-subscription of any MPI processes to cores and hence the PSCT processes were running on additional cores on Cluster B (for Cluster A, the cores were oversubscribed).

We first look at the individual benchmarks and then discuss Overheads and Scalability in Section 6.6.6.

All values were averaged over 10 runs (including the baseline case of an application without Multiverse). The speedups plot the arithmetic mean of these runs.

6.6.1 WalkSAT

WalkSAT is a randomized SAT solver that exhibits good performance. At each step the solver picks a random unsatisfied clause and “flips” a variable in the clause. We used the C based implementation of this application from [111]. Figure 52 shows the speedups that were obtained using several SAT inputs from different application domains:

- **par16-1.cnf**² and **par16-5.cnf**³ instances which arises from the problem of learning the parity function from the DIMACS benchmarks [106].
- **ii32c4.cnf**⁴ an instance from an inference problem [70].
- **frb45-21-1.cnf**⁵ and **frb45-21-2.cnf**⁶ CSP instances of Model RB used in AI [128].

Discussion We observe that large scale speculation clearly benefits WalkSAT on a variety of benchmark datasets. The benchmark datasets demonstrate good scalability and in fact the par16-5.cnf dataset (on Cluster A) and frb45-21-1.cnf (on Cluster B) show super linear speedups! Recall that this is due to the underlying distribution of execution times (discussed in Section 6.1.1). The speedups vary from dataset to dataset and are a characteristic of the underlying distribution of execution times

²1015 variables and 3310 clauses

³1015 variables and 3358 clauses

⁴759 variables and 20862 clauses

⁵945 variables and 61855 clauses

⁶945 variables and 61855 clauses

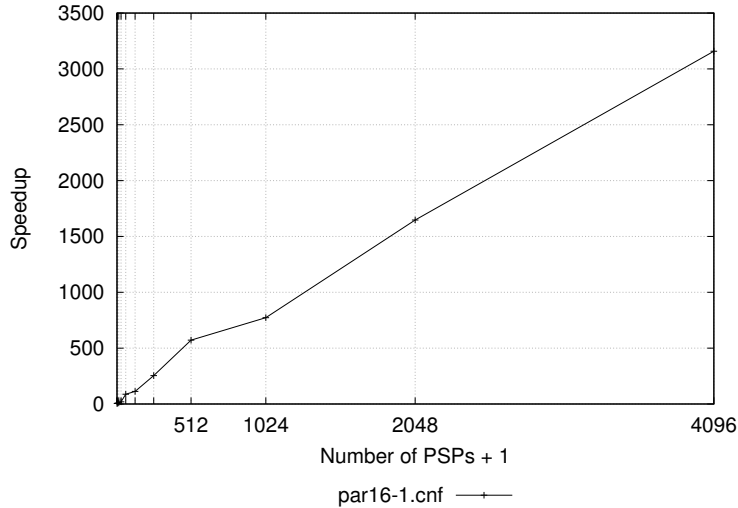


Figure 53: Results for frb45-21-2.cnf with a larger number of cores

(different datasets have different CDF plots like Figure 43(b)). We note that all these datasets exhibit good scalability going well beyond the number of cores available on a single machine. Pointing to the fact, that the use of Multiverse is crucial in extracting maximal speedup.

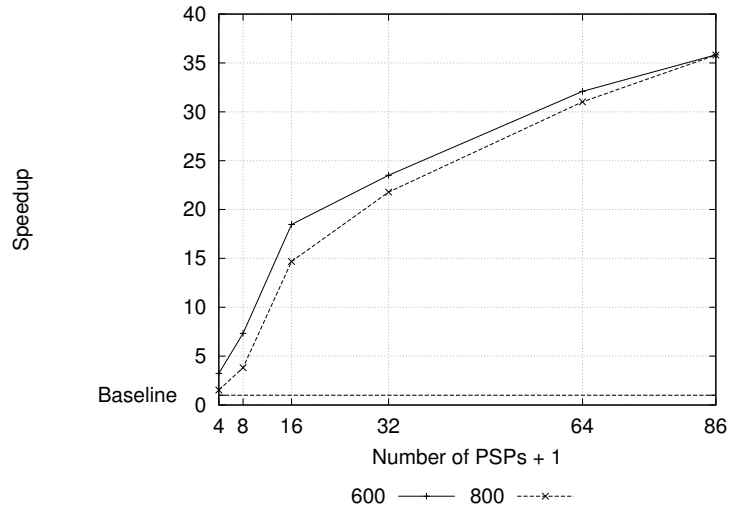
We were able to secure resources to run just the frb45-21-2.cnf dataset with a much larger number of cores. Figure 53 reports the results we obtained with approximately 4 times more PSPs (these extra resources were obtained from a larger pool of machines in Cluster B itself). We notice that this dataset is able to scale all the way up to 4096 cores.

The next three benchmarks use a multi-walk method where multiple concurrent explorations of the search space are performed starting from different initial configurations. An adaptive search framework is used [20]. The framework makes use of a short term adaptive memory to avoid getting stuck in loops and local minimas.

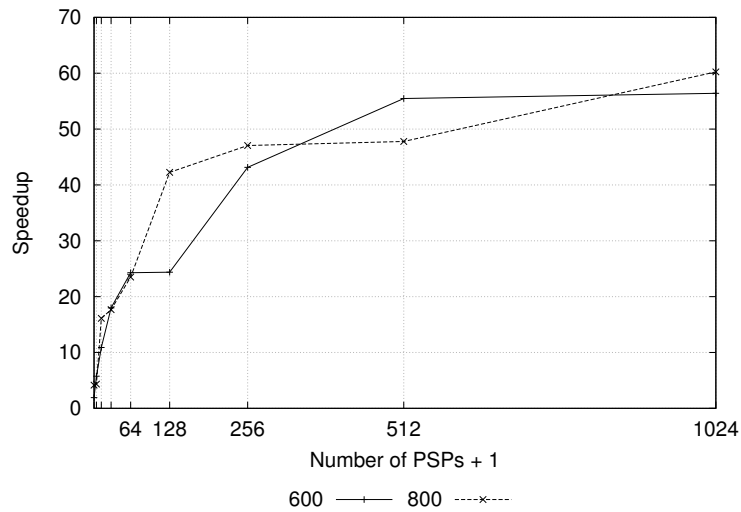
6.6.2 All Interval Series Problem

This problem is inspired by a well-known problem occurring in serial musical composition and was first used in [58]. It can be stated as: given the twelve standard pitch-classes in music ($c, c\#, d, \dots$), represented by numbers $0, 1, \dots, 11$, find a series in

which each pitch-class occurs exactly once and in which the musical intervals between neighboring notes cover the full set of intervals from the minor second (1 semitone) to the major seventh (11 semitones). We used the C based implementation of this application from [35]. Figure 54 shows the speedups we were able to obtain using an increasing number of cores for two different problem sizes.



(a) Cluster A



(b) Cluster B

Figure 54: Results from the All Interval Series Benchmark

Discussion For both problem sizes (600 and 800) we see similar speedups, while not super linear as in the case of WalkSAT these examples also show that we can achieve

very reasonable performance gains with an increasing number of cores up to 256. For these two problem sizes we observe no sizable gain in performance going beyond 256 cores. This simply means we have exhausted the amount of parallelism we can gain. Larger problems will still benefit well beyond this limit, however these larger problems simply take too long to run with a smaller number of cores (1 - 64) and hence do not lend themselves easily to speedup measurements. This example starkly indicates that improvements will taper off after a certain core count for smaller problems. We refer interested readers to prior work [28] for techniques on how to automatically determine the right degree of speculation for such problems.

6.6.3 Costas Arrays

A costas array can be regarded as a set of n points lying on the squares of a $n * n$ checkerboard. Each row or column contains only one point and all of the $n(n - 1)/2$ displacement vectors between each pair of dots are distinct [49]. These arrays are very useful in applications such as sonar and radar. We used the C based implementation of this application from [35]. Figure 55 shows the speedups we were able to obtain using an increasing number of cores for two different problem sizes.

Discussion This benchmark again demonstrates high scalability and improvements are sizable all way up to a maximum of 1024 cores on Cluster B. The benchmark with side 21 is actually a very difficult instance and takes more than an hour and a half on average to complete and speculative parallelization brings this down to a minute and a half on Cluster A and mere seconds in Cluster B, both of which are far more acceptable.

6.6.4 Perfect Square Problem

The perfect square placement problem (also called the squared square problem) is to pack a set of squares with different integer sizes into a bigger square in such a way

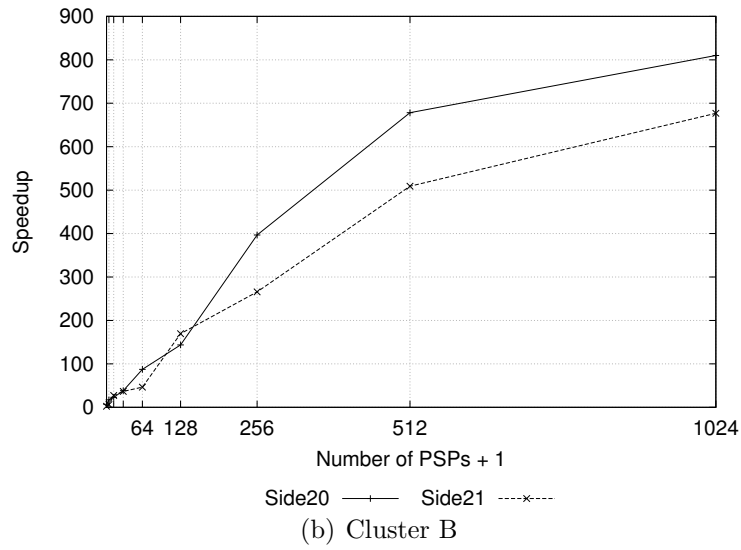
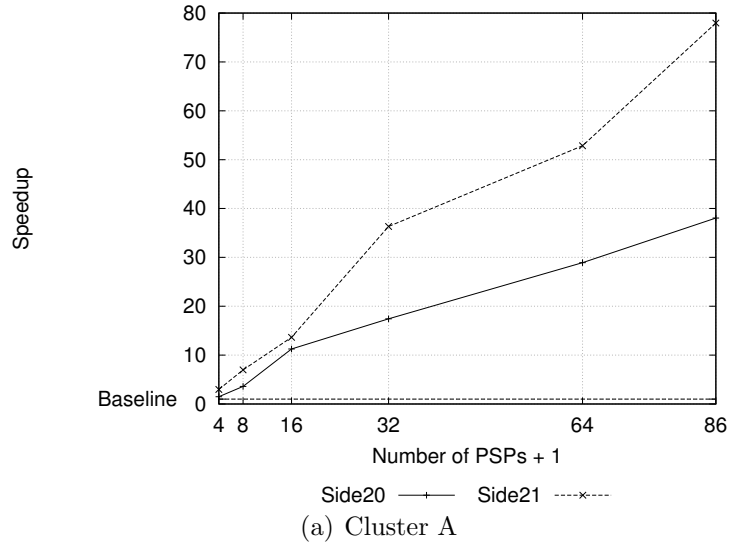


Figure 55: Results from Costas Array Benchmark

that no squares overlap each other and all square borders are parallel to the border of the big square [30]. We used the C based implementation of this application from [35]. Figure 55 shows the speedups we were able to obtain using an increasing number of cores for two different problems ⁷ ⁸.

⁷Prob1 Config: (48,25,147,(3,4,5,6,8,9,10,12,13,14,15,16,17,19,20,23,25,27,32,33,34,40,41,73,74))

⁸Prob2 Config: (49,25,208,(1,2,3,4,5,7,8,11,12,17,18,24,26,28,29,30,36,39,44,45,50,59,60,89,119))

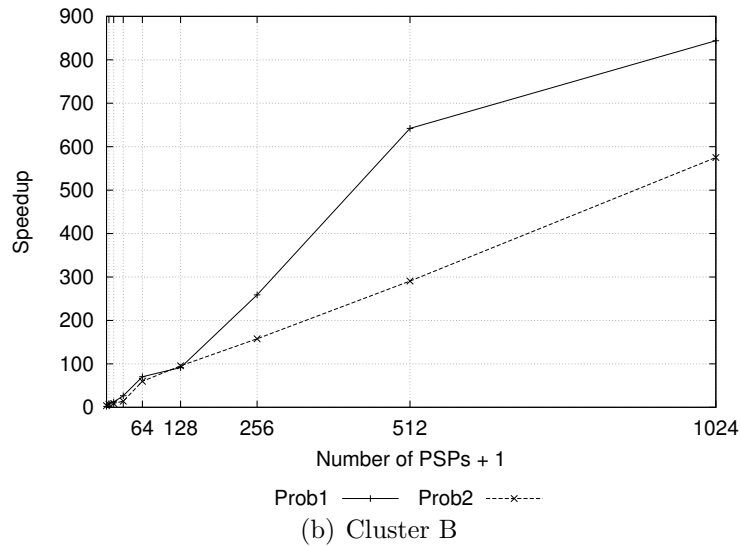
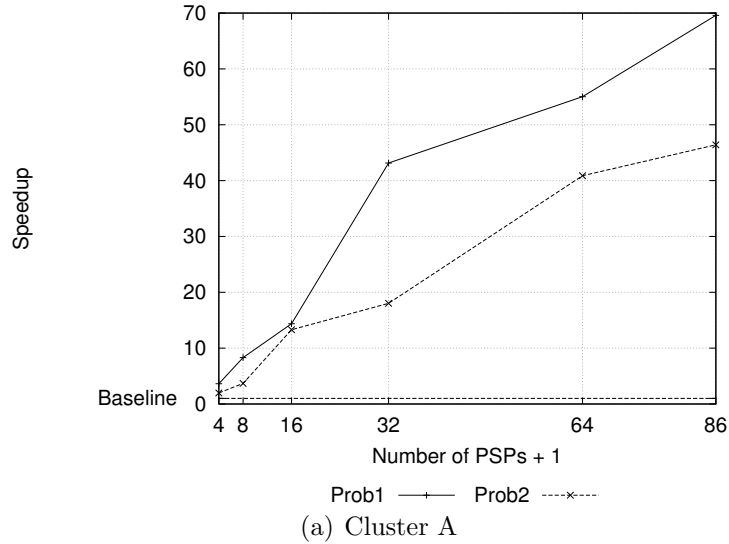


Figure 56: Results from the Perfect Square Benchmark

Discussion Similar to previous experiments we observe good speedups in the case of both problems in both the clusters. We observe different amounts of speedups between the two datasets, similar to the situation that we observed in some of the WalkSAT benchmarks.

6.6.5 Traveling Salesman Problem (TSP)

The TSP problem asks the following question: given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city

exactly once and returns to the origin city? It is an NP-hard problem in combinatorial optimization. It has many real world applications such as planning, logistics, microchip manufacturing, DNA sequencing and in operations research. We used an implementation of the Lin-Kernighan heuristic obtained from [55].

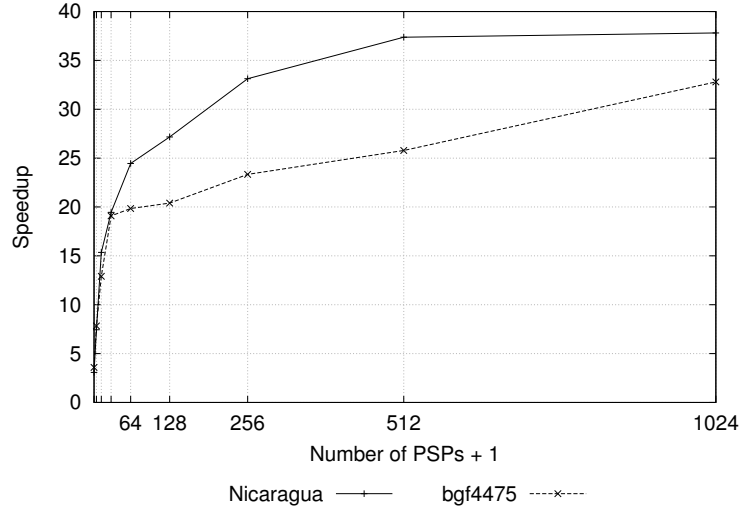


Figure 57: Results for TSP Benchmark for Cluster B

While the metric for choosing the best PSP in the previous benchmarks was execution time, for the TSP problem the Multiverse runtime is looking for a solution which is within some X units (specified below) of the optimal (each speculation updates its current quality with the runtime using an API call). Finding the optimal value is prohibitively expensive (NP-hard) so for such applications an inferior solution is commonly deemed acceptable. Cluster A was unavailable to us at the time of running these experiments, hence we report results only from Cluster B. Figure 57 shows the speedups we were able to obtain using an increasing number of cores for two different datasets:

- **Nicaragua** Containing 3,496 cities in Nicaragua derived from data from the National Imagery and Mapping Agency database of geographic feature names. It has an optimal tour of 96,132. Multiverse waits until a solution which is 96,182 units or better is found and chooses that as the best speculation.

- **bgf4475** A 4,475-city instance, derived from VLSI data from the Forschungsinstitut für Diskrete Mathematik. The optimal tour length for this instance is unknown. Multiverse waits until a solution which is 13,223 units or better is found and chooses that as the best speculation.

Discussion The bgf4475 dataset demonstrates lower speedup than Nicaragua but improves slightly with increasing core counts. The Nicaragua dataset on the other hand demonstrates improvements up to 512 cores and then performance remains almost stationary. As discussed previously, larger problems would demonstrate improvement to larger core counts and these datasets would also benefit from techniques which automatically determine the right degree of speculation (we refer to prior work [28]).

6.6.6 Overheads & Scalability

In this section we discuss overheads and scalability issues. The main overheads are during start/end of speculative regions and heap memory access during speculative regions.

The overheads at the start and end of a speculative region are fairly modest and simply involve typical collective communication across a cluster. At the start up of a speculative region the stack, execution context and some bookkeeping information needs to be sent to all the PSPs. At the end of the speculative region all PSPs are contacted, their execution is terminated and the best speculation sends its execution context, stack and some bookkeeping information back to Process0. Table 3 shows the breakup of these overheads for an application with a 4MB stack which uses 85 PSPs across the 43 machines in Cluster A and 1023 PSPs across 86 machines in in Cluster B. Table 3 confirms that a linear approach (of contacting each PSP in a serial manner) for startup and shutdown does not scale across a cluster and a tree based communication pattern decreases any overheads to minimal levels as discussed

in Section 6.5.2. The time it takes to transfer the execution context and stack back from the best speculation to Process0 does not change (between tree or linear) since it is based on a simple point to point communication. The tree based approach reduces overheads significantly and Multiverse is able to startup a speculation across Cluster A in little over half a second and Cluster B in less than 25 ms. While, the trends are similar in both clusters, Cluster B exhibits better performance since it is a HPC cluster with a much faster interconnect.

Table 3: Overheads in Cluster A and Cluster B

Approach	Tree (ms)	Linear (ms)
Startup time (A)	581.55	31522.84
Startup time (B)	22.31	3913.66
Shutdown time (A)	3.54	4.52
Shutdown time (B)	0.5268	0.8837
Best Spec. Transfer Time (A)	357.90	357.90
Best Spec. Transfer Time (B)	0.526	0.526

To observe how Multiverse behaves in a situation where a large amount of heap memory needs to be transferred, we designed a special benchmark to test Multiverse’s scalability with a larger amount of heap memory (128MB) transferred to each PSP and back. This benchmark simply allocates a large number of pages in heap memory and code in the speculative region reads and writes to all allocated pages. We plot the execution times of two versions of this benchmark. One using the Page Server Cache Tree and one without. Figure 58 reports total benchmark execution times with an increasing number of PSPs. Each page server cache in the Page Server Cache Tree serves 8 PSPs and is organized as a binary tree (default configuration).

Discussion If the Page Server Cache Tree is not used, performance quickly degrades with an increasing number of PSPs. Execution time shows an almost linear increase with number of PSPs. This is expected since Process0 becomes a bottleneck as it handles all requests for pages. By using the Page Server Cache Tree on the other hand we observe excellent scalability. The Page Server Cache Tree is suited perfectly,

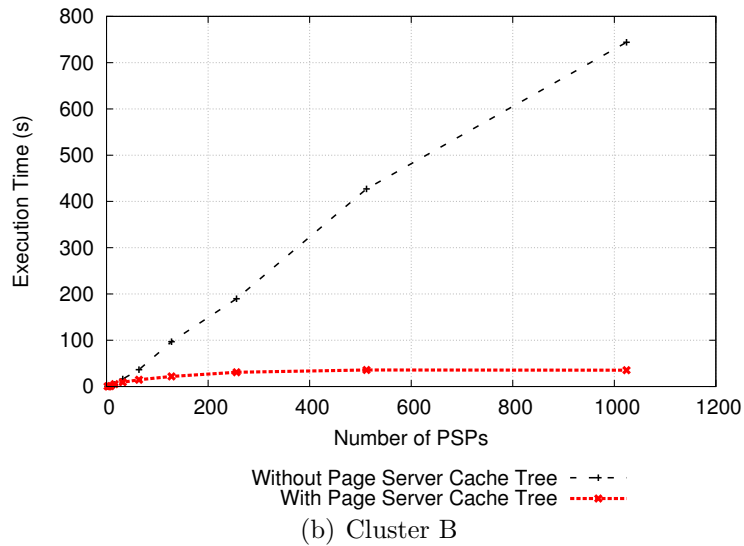
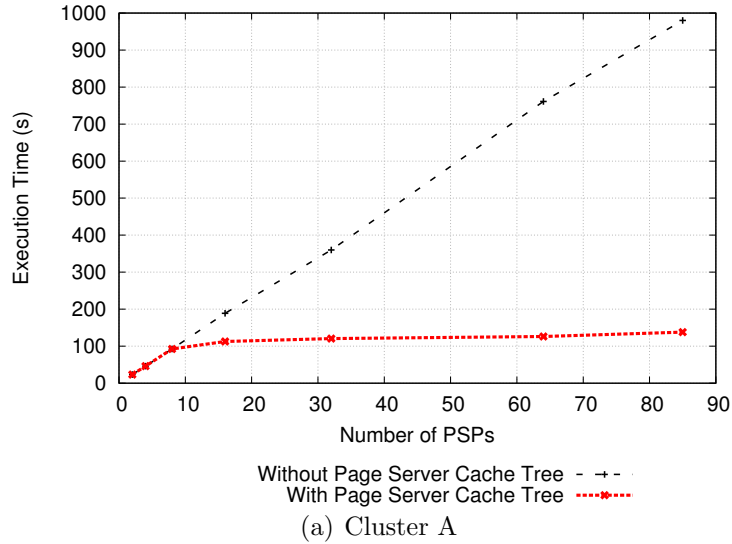


Figure 58: Multiverse Scalability

and it removes almost all scaling issues with dramatically reduced execution times increasing only in a logarithmic fashion (see Appendix A for an analysis of speedups and how they are almost ideal).

6.7 Related Work

There has been much research in use of speculation at the lower levels of execution such as in hardware or through compilers [124, 130, 53, 117].

Programmer controlled speculation is another direction of leveraging speculation.

Prabhu et al. [97] allow programmers to speculatively parallelize loops using domain specific functions to predict the value of dependencies. Such value speculation is orthogonal to our work. The Grace framework [14] allows programmers to write speculative code but focuses on using this to eliminate concurrency issues, a different use of speculation. The PetaBricks programming model [9] allows programmers to specify multiple implementations for an operation. The compiler tunes the selection of the actual implementations that are used at runtime and doesn't directly follow a speculative approach.

There has been a significant amount of interest in auto-tuning based frameworks [125, 68, 46, 77] which compare and select good candidates of code and parameters. While auto-tuning approaches are good for converging on a best parameter configuration they are not suited for exploiting applications which demonstrate a variance in execution time. Auto-tuning can be used in addition to our work.

Programming models which execute multiple variants of an algorithm or different heuristics and choose amongst them have been proposed [28, 122, 99]. Each of these models differ by the exact type of speculations they support, their ease of use, isolation guarantees and execution model. However, they are all strictly limited by the parallelism in a single machine. Multiverse makes an explicit case for large scale distributed speculation and presents novel techniques and mechanisms to deal with the non-trivial issues that come with this fundamental shift to a distributed environment.

While Multiverse expands the scope of these frameworks to the distributed setting, it does come with some limitations (see Section 6.2.1) and in that sense is somewhat less general than approaches such as CPE [122].

There is strong evidence for variance in algorithmic execution time [83, 87, 20]. Individual applications have used the idea of launching multiple instances to speed up the execution, for example ManySAT [126] and even domains as diverse as security [29] and reliability [113]. While these applications illustrate the utility of this

approach, their mechanisms are application specific and do not allow programmers to speculatively distribute computations within an existing application across a cluster.

Hosek et al. [59] use a similar idea of launching multiple versions of code in parallel in a single machine, for a different purpose, to improve reliability of Linux applications by providing safe software updates.

There are also related systems which move work onto a different machine for offloading purposes. For example, cloudlets [108] moves execution through VMs from mobile devices to close by resource rich "cloudlets". Our PSCT resembles CDNs [39] used in clusters which serve as caches for content in large scale web clusters.

Our approach of transferring execution state across processes bears similarity to that of PM2 [10], which also inspired the Charm++ implementation [69]. PM2 and Charm++ use this ability to migrate tasks exclusively for purposes such as load balancing and fault tolerance. It also bears similarity to a mechanism proposed by Rogers et al. [103, 22] however their programming model aims to migrate executions for locality purposes (further, their approach also imposes an additional restriction that programmers do not use pointers to the stack, which our model supports). Multiverse on the other hand uses a similar idea to enable speculative parallelization of applications, which has different requirements due to its one-to-many process fan out. It incorporates an on-demand component and is designed to deal with scalability issues for large scale speculation.

6.8 Summary

In this chapter we discuss how algorithmic speculation helps programmers exploit deeper algorithmic properties to extract parallelism and how semantics of many applications allow us to extract large amounts of parallelism. We presented the Multiverse programming model which can be used to easily write large scale distributed

speculations. We motivated and discussed the need for large scale distributed speculation and the challenges it involves both in terms of scalability and ease of use. We present special mechanisms to transparently and efficiently allow code to continue execution in speculative processes even on different machines (allows use of pointers). We deal with potential bottlenecks in scalability and introduced the page server cache tree. We implemented our contributions in an easy to use C library and demonstrate the Multiverse framework on a number of applications. We report significant performance improvements over several benchmarks. Large scale distributed algorithmic speculation can be used to speedup many applications and we hope that Multiverse's ease of use and special scalability techniques enable this.

CHAPTER VII

CONCLUSION

With the increasing amounts of parallelism that is available to programmers at the hardware level there is a rising burden on programmers to utilize all of these resources. Speculation often leads to an easier and more practical path to adding parallelism to applications, a golden bridge for a quick expression of "potential" parallelism and brings a refreshingly orthogonal view to the problem. While the transactional model allows programmers to express speculation, it does not deal with the semantics of speculation. In this thesis we take a multi-faceted view of the problem of speculation through a combination of programming models, compiler analysis, scheduling and runtime systems and tackle the semantic issues that surround speculation such as the degree of speculation, wasted work and rollbacks, guidance of speculation, repeatability and scalability.

While the transactional construct has helped write speculative parallel code more easily, choosing the right degree of speculation is still a difficult task. Sometimes applications (their algorithms and data structures) can simply not support the amount of parallelism that larger multi-cores that are becoming available provide. In these situations it is imperative that we do not move the burden of tuning the program on to the programmer but rather have transparent solutions which can *automatically select and adapt the right degree of speculation* and parallelism to use. Our work, F2C2-STM is able to effectively control the amount of speculation in these applications and eases the burden on the programmer significantly.

While, controlling the degree of speculation can provide sizable performance gains with no programmer effort alternate ways of exposing the semantic meaning of the

speculation can mitigate the cost of misspeculation. In some applications the semantic properties of the algorithm can be leveraged and the work performed by incorrect speculations does not have to be completely thrown away but can actually be merged with another transaction. This frees the programmer from having to worry about the performance impact of mis-speculations since the underlying system can simply salvage the work. Our *merge construct* can be used by programmers to salvage work and improve performance.

Providing programmers with the necessary tools to express semantics of the dynamic data-footprints of speculative operations through probabilistic hints helps as well. While speculation does entail speculating "incorrectly" at times, for best performance these instances should be minimized. There are often semantic properties of the data structures and algorithms that can help *guide speculations*. We show how the programmer can easily expose these properties and how it can significantly improve overall performance.

While, writing speculative parallel applications using the transactional paradigm has helped side-step many of the issues with traditional approaches it is still not a panacea for all bugs. It is still possible to make mistakes while using the transactional construct. The very nature of speculation with aborts, rollbacks and retries can make the debugging process more difficult. STMs have removed many of the concerns of parallel programming but a key remaining challenge which complicates speculative application development is non-determinism. Our work with DeSTM aims to remove this limitation and aids programmers by introducing *determinism into the development cycle*. Programmers are very used to developing code assuming it is deterministic. Non-determinism throws a wrench in the iterative development process. We demonstrate how we can introduce determinism to significantly improve the development experience without undue performance loss.

Deeper algorithmic properties such as variance and alternative implementation

options provide an abundant source of parallelism. In resource rich environments where execution time is crucial exploiting these avenues can provide for significant scalable performance improvements. Our Multiverse programming model efficiently supports *distributed algorithmic speculations* allowing programmers to quickly harness this kind of parallelism in their applications efficiently.

As increasing parallel resources become more common, programmers need to adapt to this new programming landscape. Speculation often leads to an easier and more practical path to adding parallelism to applications. The proposed solutions from this work tackle the semantic issues surrounding speculative parallelism from various angles and demonstrate how we can significantly improve the performance of coarse grain speculation while at the same time making it even easier to use from the programmers perspective laying the foundation for the development and tuning of new speculative algorithms. Especially as the amount of parallelism continuously increases such techniques are vital in effectively using the resources that are available.

7.1 Future Work

Many of the topics investigated in this thesis have extensions which are good starting points for future exploration.

An interesting avenue of exploration when leveraging the Multiverse framework with parallel applications is the trade-off between speculation and parallelism. Consider the situation in which each of the individual speculations are themselves parallel. Each of these could be parallelized using traditional parallelization techniques such as data parallelization. Now, a question arises in how much resources to give to each of these individual parallel executions vs. how much to give to the overall speculation. The answer is not straight forward. While, speculative parallelization increases the diversity of executions increasing the parallel resources to each individual execution increases the speed at which it can process a certain portion of the data set. A smart

runtime mechanism can help monitor the progress in these and dynamically balance the parallelism between them towards achieving the best performance.

Incorporating predictive information into runtimes which manage speculative execution appears to be a potential solution to further increase performance. Consider a predictive scheme which is capable of predicting how long it will take to start executing a transaction in an STM application. Let's assume that a predictive scheme can warn of a transaction say X ms before it actually occurs. Now, if all parallel threads are continuously predicting when transactional regions are going to be executed we can use this information to make several improvements to transaction scheduling. Firstly, if we notice that these predictions do not overlap with one another then we can in fact revert to traditional synchronization primitives such as locks instead of incurring the transactional overhead with speculative executions. Secondly, if we know the probabilities of conflicts between pairs of transactions, highly conflicting pairs can be scheduled such that they do not overlap based on this predictive information.

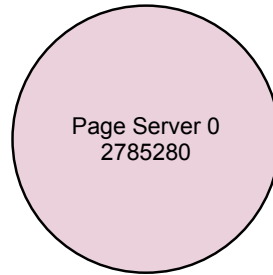
Overall, with the use of speculative parallelism increasing, innovative ideas and techniques are required to make effective use of the parallel resources that are available.

APPENDIX A

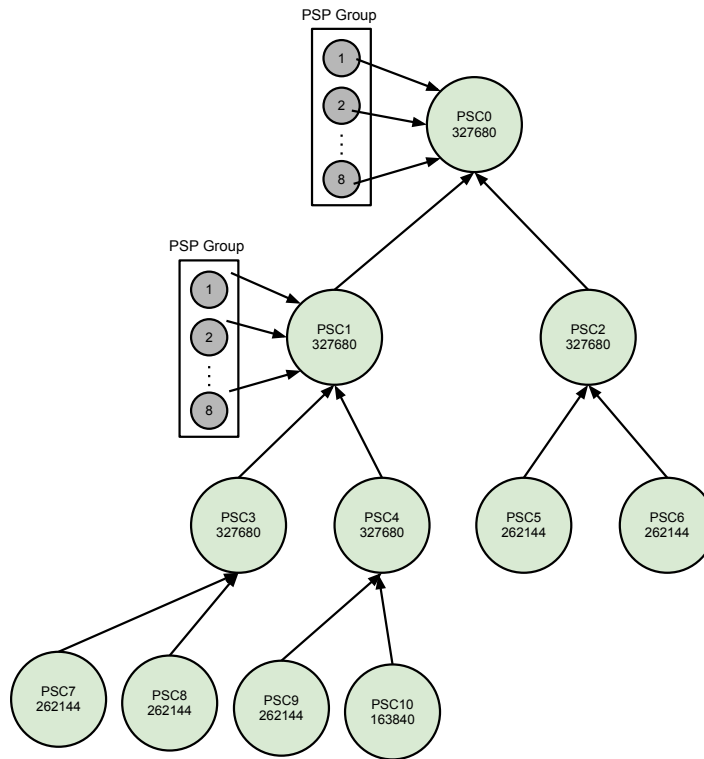
PAGE ACCESS COUNTS

Consider the experiment in Section 6.6.6 which tested the scalability of Multiverse (see Figure 58(a)) on cluster A. The experiment transferred 128MB of heap memory between Process0 and the 85 PSPs. That is equivalent to 32,768 pages of memory to each PSP (using a 4KB page size). If the Page Server Cache Tree (PSCT) is not used requests for every page of memory goes to the Page Server on Process0 and it ends up having to reply to 2,785,280 requests (Figure 59(a)). This is a gigantic burden on Process0 and the performance in Figure 58 reflects this (as expected).

However, with the use of the PSCT we can dramatically decrease the number of requests that need to be answered by any single process. Figure 59(b) shows how the number of accesses split up in the case of 85 PSPs. Here the PSCT is configured as a binary tree and each Page Server Cache (PSC) serves 8 PSPs (only 2 of the 11 PSP groups are shown in Figure 59(b)). That means at max each PSC replies to requests from 10 processes (8 PSPs and 2 children PSCs). That means at max, each one would serve 32,768 pages for 10 processes. A maximum of 327,680 pages compared to the maximum of 2,785,280 in the case of Figure 59(a) an 88% reduction in number of requests that need to be served. In the experiment (Figure 58) we observe an 86% reduction in execution time (with and without PSCT) demonstrating that fact that the PSCT is able to provide excellent scalability thereby allowing Multiverse to provide performance benefits close to the theoretical limit.



(a) Without the Page Server Cache Tree



(b) With the Page Server Cache Tree

Figure 59: Page Access Counts

REFERENCES

- [1] ABDEL-SHAFI, H., SPEIGHT, E., and BENNETT, J. K., “Efficient user-level thread migration and checkpointing on windows nt clusters,” in *Proceedings of the 3rd conference on USENIX Windows NT Symposium - Volume 3*, WINSYM’99, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 1999.
- [2] AFEK, Y., KORLAND, G., and ZILBERSTEIN, A., “Lowering stm overhead with static analysis,” in *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, LCPC’10, (Berlin, Heidelberg), pp. 31–45, Springer-Verlag, 2011.
- [3] AGGARWAL, A. and ANDERSON, R., “A random nc algorithm for depth first search,” in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC ’87, (New York, NY, USA), pp. 325–334, ACM, 1987.
- [4] AGGARWAL, A., ANDERSON, R. J., and KAO, M.-Y., “Parallel depth-first search in general directed graphs,” in *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, STOC ’89, (New York, NY, USA), pp. 297–308, ACM, 1989.
- [5] ALLEN, M. D., SRIDHARAN, S., and SOHI, G. S., “Serialization sets: a dynamic dependence-based parallel execution model,” in *PPoPP ’09*, (New York, NY, USA), pp. 85–96, ACM, 2009.
- [6] ANSARI, M., KOTSELIDIS, C., JARVIS, K., LUJÁN, M., KIRKHAM, C., and WATSON, I., “Advanced concurrency control for transactional memory using transaction commit rate,” in *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Euro-Par ’08, (Berlin, Heidelberg), pp. 719–728, Springer-Verlag, 2008.
- [7] ANSARI, M., LUJÁN, M., KOTSELIDIS, C., JARVIS, K., KIRKHAM, C., and WATSON, I., “Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering,” in *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC ’09, (Berlin, Heidelberg), pp. 4–18, Springer-Verlag, 2009.
- [8] ANSARI, M., LUJÁN, M., KOTSELIDIS, C., JARVIS, K., KIRKHAM, C., and WATSON, I., “Transactions on high-performance embedded architectures and compilers iii,” ch. Robust Adaptation to Available Parallelism in Transactional Memory Applications, pp. 236–255, Berlin, Heidelberg: Springer-Verlag, 2011.
- [9] ANSEL, J., CHAN, C., WONG, Y. L., OLSZEWSKI, M., ZHAO, Q., EDELMAN, A., and AMARASINGHE, S., “Petabricks: a language and compiler for

- algorithmic choice,” in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, (New York, NY, USA), pp. 38–49, ACM, 2009.
- [10] ANTONIU, G., BOUGÉ, L., and NAMYST, R., “An efficient and transparent thread migration scheme in the pm2 runtime system,” in *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, (London, UK, UK), pp. 496–510, Springer-Verlag, 1999.
- [11] AVIRAM, A., WENG, S.-C., HU, S., and FORD, B., “Efficient system-enforced deterministic parallelism,” in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, (Berkeley, CA, USA), pp. 1–16, USENIX Association, 2010.
- [12] BADER, D. and CONG, G., “Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs,” in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, p. 39, april 2004.
- [13] BERGAN, T., ANDERSON, O., DEVIETTI, J., CEZE, L., and GROSSMAN, D., “Coredet: A compiler and runtime system for deterministic multithreaded execution,” in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, (New York, NY, USA), pp. 53–64, ACM, 2010.
- [14] BERGER, E. D., YANG, T., LIU, T., and NOVARK, G., “Grace: Safe multithreaded programming for c/c++,” in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, (New York, NY, USA), pp. 81–96, ACM, 2009.
- [15] BEVILACQUA, A., LANZA, A., BACCARANI, G., and ROVATTI, R., “A single-scan algorithm for connected components labelling in a traffic monitoring application,” in *Proceedings of the 13th Scandinavian conference on Image analysis*, SCIA'03, (Berlin, Heidelberg), pp. 677–684, Springer-Verlag, 2003.
- [16] BIENIUSA, A., MIDDELKOOP, A., and THIEMANN, P., “Brief announcement: actions in the twilight - concurrent irrevocable transactions and inconsistency repair,” in *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, (New York, NY, USA), pp. 71–72, ACM, 2010.
- [17] BLELLOCH, G. E., FINEMAN, J. T., GIBBONS, P. B., and SHUN, J., “Internally deterministic parallel algorithms can be fast,” in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '12, (New York, NY, USA), pp. 181–192, ACM, 2012.
- [18] BOCCHINO, JR., R. L., ADVE, V. S., DIG, D., ADVE, S. V., HEUMANN, S., KOMURAVELLI, R., OVERBEY, J., SIMMONS, P., SUNG, H., and VAKILIAN,

- M., “A type and effect system for deterministic parallel java,” in *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, (New York, NY, USA), pp. 97–116, ACM, 2009.
- [19] BOCCHINO, JR., R. L., ADVE, V. S., DIG, D., ADVE, S. V., HEUMANN, S., KOMURAVELLI, R., OVERBEY, J., SIMMONS, P., SUNG, H., and VAKILIAN, M., “A type and effect system for deterministic parallel java,” in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, (New York, NY, USA), pp. 97–116, ACM, 2009.
- [20] CANIOU, Y., DIAZ, D., RICHOUX, F., CODOGNET, P., and ABREU, S., “Performance analysis of parallel constraint-based local search,” in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPOPP '12, (New York, NY, USA), pp. 337–338, ACM, 2012.
- [21] CAO MINH, C., CHUNG, J., KOZYRAKIS, C., and OLUKOTUN, K., “STAMP: Stanford transactional applications for multi-processing,” in *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.
- [22] CARLISLE, M. C. and ROGERS, A., “Software caching and computation migration in olden,” in *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, (New York, NY, USA), pp. 29–38, ACM, 1995.
- [23] CARRILLO, J. F., ORKISZ, M., and HOYOS, M. H., “Extraction of 3d vascular tree skeletons based on the analysis of connected components evolution,” in *Proceedings of the 11th international conference on Computer Analysis of Images and Patterns*, CAIP'05, (Berlin, Heidelberg), pp. 604–611, Springer-Verlag, 2005.
- [24] CHAN, K., LAM, K. T., and LI WANG, C., “Adaptive thread scheduling techniques for improving scalability of software transactional memory,” in *Proceedings of the 10th IASTED-PDCN, 2011*, p. 91-98.
- [25] CHENG, G.-I., FENG, M., LEISERSON, C. E., RANDALL, K. H., and STARK, A. F., “Detecting data races in cilk programs that use locks,” in *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, (New York, NY, USA), pp. 298–309, ACM, 1998.
- [26] CHENG, H.-Y., LIN, C.-H., LI, J., and YANG, C.-L., “Memory latency reduction via thread throttling,” in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 53–64, IEEE Computer Society, 2010.

- [27] CLEDAT, R., RAVICHANDRAN, K., and PANDE, S., “Leveraging data-structure semantics for efficient algorithmic parallelism,” in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF ’11, (New York, NY, USA), pp. 28:1–28:10, ACM, 2011.
- [28] CLEDAT, R. E., KUMAR, T., and PANDE, S., “Efficiently speeding up sequential computation through the n-way programming model,” in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA ’11, (New York, NY, USA), pp. 537–554, ACM, 2011.
- [29] COX, B., EVANS, D., FILIPI, A., ROWANHILL, J., HU, W., DAVIDSON, J., KNIGHT, J., NGUYEN-TUONG, A., and HISER, J., “N-variant systems: a secretless framework for security through diversity,” in *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS’06, (Berkeley, CA, USA), USENIX Association, 2006.
- [30] CROFT, H., FALCONER, K., and GUY, R., *Unsolved problems in geometry*. Problem books in mathematics, Springer-Verlag, 1991.
- [31] DALESSANDRO, L., SPEAR, M. F., and SCOTT, M. L., “Norec: Streamlining stm by abolishing ownership records,” in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP ’10, (New York, NY, USA), pp. 67–78, ACM, 2010.
- [32] DEVI, S. M. R. and BHAGVATI, C., “Connected component in feature space to capture high level semantics in cbir,” in *Proceedings of the Fourth Annual ACM Bangalore Conference*, COMPUTE ’11, (New York, NY, USA), pp. 5:1–5:6, ACM, 2011.
- [33] DEVIETTI, J., LUCIA, B., CEZE, L., and OSKIN, M., “Dmp: Deterministic shared memory multiprocessing,” in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, (New York, NY, USA), pp. 85–96, ACM, 2009.
- [34] DEVIETTI, J., NELSON, J., BERGAN, T., CEZE, L., and GROSSMAN, D., “Rcdc: A relaxed consistency deterministic computer,” in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, (New York, NY, USA), pp. 67–78, ACM, 2011.
- [35] DIAZ, D., CODOGNET, P., and ABREU, S., “Adaptive search distribution, <http://cri-dist.univ-paris1.fr/diaz/adaptive/>.”
- [36] DICE, D., SHALEV, O., and SHAVIT, N., “Transactional locking ii,” in *Proceedings of the 20th International Conference on Distributed Computing*, DISC’06, (Berlin, Heidelberg), pp. 194–208, Springer-Verlag, 2006.

- [37] DIDONA, D., FELBER, P., HARMANCI, D., ROMANO, P., and SCHENKER, J., “Identifying the optimal level of parallelism in transactional memory applications,” in *Networked Systems* (GRAMOLI, V. and GUERRAOU, R., eds.), vol. 7853 of *Lecture Notes in Computer Science*, pp. 233–247, Springer Berlin Heidelberg, 2013. Related presentation: http://www.lip6.fr/public/2012-09-27_Felber.pdf.
- [38] DIETER, W. R. and LUMPP, JR., J. E., “User-level checkpointing for linuxthreads programs,” in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 81–92, USENIX Association, 2001.
- [39] DILLEY, J., MAGGS, B., PARIKH, J., PROKOP, H., SITARAMAN, R., and WEIHL, B., “Globally distributed content delivery,” *IEEE Internet Computing*, vol. 6, pp. 50–58, Sept. 2002.
- [40] DOLEV, S., HENDLER, D., and SUISSA, A., “Car-stm: Scheduling-based collision avoidance and resolution for software transactional memory,” in *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC ’08, (New York, NY, USA), pp. 125–134, ACM, 2008.
- [41] DRAGOJEVIĆ, A., GUERRAOU, R., SINGH, A. V., and SINGH, V., “Preventing versus curing: Avoiding conflicts in transactional memories,” in *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC ’09, (New York, NY, USA), pp. 7–16, ACM, 2009.
- [42] FELBER, P., FETZER, C., and RIEGEL, T., “Dynamic performance tuning of word-based software transactional memory,” in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP ’08, (New York, NY, USA), pp. 237–246, ACM, 2008.
- [43] FLEISCHER, L., HENDRICKSON, B., and PINAR, A., “On identifying strongly connected components in parallel,” in *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, IPDPS ’00, (London, UK, UK), pp. 505–511, Springer-Verlag, 2000.
- [44] FREEMAN, J., “Parallel algorithms for depth-first search,” Technical Report (CIS) MS-CIS-91-71, University of Pennsylvania, University of Pennsylvania, Philadelphia, PA 19104-6389, October 1991.
- [45] FRIGO, M., LEISERSON, C. E., and RANDALL, K. H., “The implementation of the cilk-5 multithreaded language,” in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI ’98, (New York, NY, USA), pp. 212–223, ACM, 1998.
- [46] FURSIN, G., COHEN, A., O’BOYLE, M., and TEMAM, O., “A practical method for quickly evaluating program optimizations,” in *Proceedings of the First international conference on High Performance Embedded Architectures*

- and Compilers*, HiPEAC'05, (Berlin, Heidelberg), pp. 29–46, Springer-Verlag, 2005.
- [47] GARG, R. P. and SHARAPOV, I., *Techniques for Optimizing Applications: High Performance Computing*. Prentice Hall Professional Technical Reference, 2002.
- [48] GIBBONS, P. B., “A more practical pram model,” in *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '89, (New York, NY, USA), pp. 158–168, ACM, 1989.
- [49] GOLOMB, S. and TAYLOR, H., “Constructions and properties of costas arrays,” *Proceedings of the IEEE*, vol. 72, no. 9, pp. 1143–1163, 1984.
- [50] GOTTSCHLICH, J. E., HERLIHY, M. P., POKAM, G. A., and SIEK, J. G., “Visualizing transactional memory,” in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, (New York, NY, USA), pp. 159–170, ACM, 2012.
- [51] GOTTSCHLICH, J. E., KNAUERHASE, R., and POKAM, G., “But how do we really debug transactional memory programs?,” in *Presented as part of the 5th USENIX Workshop on Hot Topics in Parallelism*, (Berkeley, CA), USENIX, 2013.
- [52] GOTTSCHLICH, J. E., VACHHARAJANI, M., and SIEK, J. G., “An efficient software transactional memory using commit-time invalidation,” in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, (New York, NY, USA), pp. 101–110, ACM, 2010.
- [53] HAMMOND, L., WILLEY, M., and OLUKOTUN, K., “Data speculation support for a chip multiprocessor,” in *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS VIII, (New York, NY, USA), pp. 58–69, ACM, 1998.
- [54] HARRIS, T., LARUS, J., and RAJWAR, R., *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd ed., 2010.
- [55] HELSGAUN, K., “An effective implementation of the linkernighan traveling salesman heuristic,” *European Journal of Operational Research*, vol. 126, no. 1, pp. 106 – 130, 2000.
- [56] HERLIHY, M. and KOSKINEN, E., “Transactional boosting: a methodology for highly-concurrent transactional objects,” in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPOPP '08, (New York, NY, USA), pp. 207–216, ACM, 2008.
- [57] HIRSCHBERG, D., “A parallel graph algorithm for finding connected components,” Technical Report TR7518, Rice University, ECE, October 1975.

- [58] HOOS, H., *Stochastic Local Search - Methods, Models, Applications*. IOS Press.
- [59] HOSEK, P. and CADAR, C., “Safe software updates via multi-version execution,” in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE ’13, (Piscataway, NJ, USA), pp. 612–621, IEEE Press, 2013.
- [60] HOWER, D. R., DUDNIK, P., HILL, M. D., and WOOD, D. A., “Calvin: Deterministic or not? free will to choose,” in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA ’11, (Washington, DC, USA), pp. 333–334, IEEE Computer Society, 2011.
- [61] IANCU, C., HOFMEYR, S., BLAGOJEVIC, F., and ZHENG, Y., “Oversubscription on multicore processors,” in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–11, April 2010.
- [62] IEEE, T. and GROUP, T. O., “The open group base specifications, issue 6, ieee std 1003.1,” 2004.
- [63] III, W. N. S. and SCOTT, M. L., “Contention management in dynamic software transactional memory,” 2004.
- [64] IN GCC, T. M., “<http://gcc.gnu.org/wiki/transactionalmemory>,”
- [65] IN HASWELL, T. S., “<https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell>,”
- [66] JAU, U. L. and TEH, C. S., “Real-time object-based video segmentation using colour segmentation and connected component labeling,” in *Proceedings of the 1st International Visual Informatics Conference on Visual Informatics: Bridging Research and Practice*, IVIC ’09, (Berlin, Heidelberg), pp. 110–121, Springer-Verlag, 2009.
- [67] JOHNSON, D. B. and METAXAS, P., “A parallel algorithm for computing minimum spanning trees,” in *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, SPAA ’92, (New York, NY, USA), pp. 363–372, ACM, 1992.
- [68] JUNG, C., LIM, D., LEE, J., and HAN, S., “Adaptive execution techniques for smt multiprocessor architectures,” in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP ’05, (New York, NY, USA), pp. 236–246, ACM, 2005.
- [69] KALE, L. V. and KRISHNAN, S., “Charm++: a portable concurrent object oriented system based on c++,” in *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA ’93, (New York, NY, USA), pp. 91–108, ACM, 1993.

- [70] KAMATH, A. P., KARMARKAR, N. K., RAMAKRISHNAN, K. G., and RESENDE, M. G. C., “A continuous approach to inductive inference,” *Math. Program.*, vol. 57, pp. 215–238, Nov. 1992.
- [71] KANG, S. and BADER, D. A., “An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs,” in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '09, (New York, NY, USA), pp. 15–24, ACM, 2009.
- [72] KARP, R. M. and MILLER, R. E., “Parallel program schemata,” *J. Comput. Syst. Sci.*, vol. 3, pp. 147–195, May 1969.
- [73] KORL, G., SHAVIT, N., and FELBER, P., “Noninvasive concurrency with java stm.”
- [74] KULKARNI, M., PINGALI, K., WALTER, B., RAMANARAYANAN, G., BALA, K., and CHEW, L. P., “Optimistic parallelism requires abstractions,” in *PLDI '07*, pp. 211–222, 2007.
- [75] KULKARNI, M., PINGALI, K., WALTER, B., RAMANARAYANAN, G., BALA, K., and CHEW, L. P., “Optimistic parallelism requires abstractions,” in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, (New York, NY, USA), pp. 211–222, ACM, 2007.
- [76] LARUS, J. R. and RAJWAR, R., *Transactional Memory*. Morgan and Claypool, 2006.
- [77] LAU, J., ARNOLD, M., HIND, M., and CALDER, B., “Online performance auditing: using hot optimizations without getting burned,” in *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, (New York, NY, USA), pp. 239–251, ACM, 2006.
- [78] LEBLANC, T. J. and MELLOR-CRUMMEY, J. M., “Monitoring and debugging of distributed real-time systems,” ch. Debugging Parallel Programs with Instant Replay, pp. 301–311, Los Alamitos, CA, USA: IEEE Computer Society Press, 1995.
- [79] LEE, J., PARK, J.-H., KIM, H., JUNG, C., LIM, D., and HAN, S., “Adaptive execution techniques of parallel programs for multiprocessors,” *J. Parallel Distrib. Comput.*, vol. 70, pp. 467–480, May 2010.
- [80] LEUNG, K.-C., CHEN, Y., and HUANG, Z., “Restricted admission control in view-oriented transactional memory,” *J. Supercomput.*, vol. 63, pp. 348–366, Feb. 2013.
- [81] LIU, T., CURTSINGER, C., and BERGER, E. D., “Dthreads: Efficient deterministic multithreading,” in *Proceedings of the Twenty-Third ACM Symposium*

- on *Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 327–336, ACM, 2011.
- [82] LU, K., ZHOU, X., BERGAN, T., and WANG, X., “Efficient deterministic multithreading without global barriers,” in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '14, (New York, NY, USA), pp. 287–300, ACM, 2014.
- [83] LUBY, M. and ERTEL, W., “Optimal parallelization of las vegas algorithms,” in *Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '94, (London, UK, UK), pp. 463–474, Springer-Verlag, 1994.
- [84] MALDONADO, W., MARLIER, P., FELBER, P., SUISSA, A., HENDLER, D., FEDOROVA, A., LAWALL, J. L., and MULLER, G., “Scheduling support for transactional memory contention management,” in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '10, (New York, NY, USA), pp. 79–90, ACM, 2010.
- [85] MENDEZ-LOJO, M., NGUYEN, D., PROUNTZOS, D., SUI, X., HASSAN, M. A., KULKARNI, M., BURTSCHER, M., and PINGALI, K., “Structure-driven optimization for amorphous data-parallel programs,” in *PPOPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, (New York, NY, USA), ACM, 2010.
- [86] MERRIFIELD, T. and ERIKSSON, J., “Conversion: Multi-version concurrency control for main memory segments,” in *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, (New York, NY, USA), pp. 127–139, ACM, 2013.
- [87] MITZENMACHER, M. and UPFAL, E., *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.
- [88] MONTESINOS, P., CEZE, L., and TORRELLAS, J., “Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently,” in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, (Washington, DC, USA), pp. 289–300, IEEE Computer Society, 2008.
- [89] MUKHERJEE, S. S., SHARMA, S. D., HILL, M. D., LARUS, J. R., ROGERS, A., and SALTZ, J., “Efficient support for irregular applications on distributed-memory machines,” in *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, (New York, NY, USA), pp. 68–79, ACM, 1995.
- [90] NARAYANASAMY, S., PEREIRA, C., and CALDER, B., “Recording shared memory dependencies using strata,” in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, (New York, NY, USA), pp. 229–240, ACM, 2006.

- [91] NEGARA, S., ZHENG, G., PAN, K.-C., NEGARA, N., JOHNSON, R. E., KALÉ, L. V., and RICKER, P. M., “Automatic mpi to ampi program transformation using photran,” in *Proceedings of the 2010 conference on Parallel processing, Euro-Par 2010*, (Berlin, Heidelberg), pp. 531–539, Springer-Verlag, 2011.
- [92] “AMD64 Architecture Programmers Manual, Volume 2,” 2010.
- [93] OLSZEWSKI, M., ANSEL, J., and AMARASINGHE, S., “Kendo: Efficient deterministic multithreading in software,” in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIV*, (New York, NY, USA), pp. 97–108, ACM, 2009.
- [94] PACHECO, P. S., *Parallel programming with MPI*. 1996.
- [95] PATIL, S. S., “Record of the project mac conference on concurrent systems and parallel computation,” ch. Closure Properties of Interconnections of Determinate Systems, pp. 107–116, New York, NY, USA: ACM, 1970.
- [96] PERELMAN, D., BYSHEVSKY, A., LITMANOVICH, O., and KEIDAR, I., “Smv: Selective multi-versioning stm,” in *Proceedings of the 25th International Conference on Distributed Computing, DISC’11*, (Berlin, Heidelberg), pp. 125–140, Springer-Verlag, 2011.
- [97] PRABHU, P., RAMALINGAM, G., and VASWANI, K., “Safe programmable speculative parallelism,” in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation, PLDI ’10*, (New York, NY, USA), pp. 50–61, ACM, 2010.
- [98] PUSUKURI, K. K., GUPTA, R., and BHUYAN, L. N., “Thread reinforcer: Dynamically determining number of threads via os level monitoring,” in *Proceedings of the 2011 IEEE International Symposium on Workload Characterization, IISWC ’11*, (Washington, DC, USA), pp. 116–125, IEEE Computer Society, 2011.
- [99] PYLA, H. K., RIBBENS, C., and VARADARAJAN, S., “Exploiting coarse-grain speculative parallelism,” in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications, OOPSLA ’11*, (New York, NY, USA), pp. 555–574, ACM, 2011.
- [100] REID, W., KELLY, W., and CRAIK, A., “Reasoning about inherent parallelism in modern object-oriented languages,” in *ACSC ’08: Proceedings of the thirty-first Australasian conference on Computer science*, (Darlinghurst, Australia, Australia), pp. 27–36, Australian Computer Society, Inc., 2008.
- [101] RIEKER, M., ANSEL, J., and COOPERMAN, G., “Transparent user-level checkpointing for the native posix thread library for linux,” in *The 2006 International Conference on Parallel and Distributed Processing Techniques and Applications*, Jun 2006.

- [102] RINARD, M. C. and LAM, M. S., “The design, implementation, and evaluation of jade,” *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 3, pp. 483–545, 1998.
- [103] ROGERS, A., CARLISLE, M. C., REPPY, J. H., and HENDREN, L. J., “Supporting dynamic data structures on distributed-memory machines,” *ACM Trans. Program. Lang. Syst.*, vol. 17, pp. 233–263, Mar. 1995.
- [104] RUGHETTI, D., DI SANZO, P., CICIANI, B., and QUAGLIA, F., “Machine learning-based self-adjusting concurrency in software transactional memory systems,” in *Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS '12*, (Washington, DC, USA), pp. 278–285, IEEE Computer Society, 2012.
- [105] RUPPERT, J., “A delaunay refinement algorithm for quality 2-dimensional mesh generation,” in *Selected Papers from the Fourth Annual ACM SIAM Symposium on Discrete Algorithms, SODA '93*, (Orlando, FL, USA), pp. 548–585, Academic Press, Inc., 1995.
- [106] RUTGERS, “Dimacs benchmarks, <http://dimacs.rutgers.edu/challenges/>.”
- [107] SANZO, P. D., RE, F. D., RUGHETTI, D., CICIANI, B., and QUAGLIA, F., “Self-optimizing concurrency in software transactional memory via model-based approach,” Tech. Rep. ISSN: 2281-4299, Department of Computer, Control, and Management Engineering, Universita di Roma, 2013.
- [108] SATYANARAYANAN, M., BAHL, P., CACERES, R., and DAVIES, N., “The case for vm-based cloudlets in mobile computing,” *IEEE Pervasive Computing*, vol. 8, pp. 14–23, Oct. 2009.
- [109] SCHEDULINGTRANSACTIONS, “<http://lpdserver.epfl.ch/transactions/wiki/doku.php?id=scheduling>,”
- [110] SCHERER, III, W. N. and SCOTT, M. L., “Advanced contention management for dynamic software transactional memory,” in *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing, PODC '05*, (New York, NY, USA), pp. 240–248, ACM, 2005.
- [111] SELMAN, B., KAUTZ, H. A., and COHEN, B., “Noise strategies for improving local search,” in *Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, AAAI '94, (Menlo Park, CA, USA), pp. 337–343, American Association for Artificial Intelligence, 1994.
- [112] SHAVIT, N. and TOUITOU, D., “Software transactional memory,” in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, PODC '95*, (New York, NY, USA), pp. 204–213, ACM, 1995.

- [113] SHYE, A., BLOMSTEDT, J., MOSELEY, T., REDDI, V. J., and CONNORS, D. A., “Plr: A software approach to transient fault tolerance for multicore architectures,” *IEEE Trans. Dependable Secur. Comput.*, vol. 6, pp. 135–148, Apr. 2009.
- [114] SMILJKOVIC, V., FETZER, C., UNSAL, O., CRISTAL, A., and VALERO, M., “Deterministic execution of tm-based applications,” 2013. Abstract: <http://www.eurotm.org/action-meetings/wtm2013/program/abstracts>, Related Report: <http://www.gsd.inesc-id.pt/~mcouceiro/eurotm/stsm/smiljkovic.pdf>, Related presentation: <http://www.gsd.inesc-id.pt/~mcouceiro/eurotm/wtm2013/smiljkovic.pdf>.
- [115] SPEAR, M. F., DALESSANDRO, L., MARATHE, V. J., and SCOTT, M. L., “A comprehensive strategy for contention management in software transactional memory,” in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP ’09, (New York, NY, USA), pp. 141–150, ACM, 2009.
- [116] STEELE, JR., G. L., “Making asynchronous parallelism safe for the world,” in *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’90, (New York, NY, USA), pp. 218–231, ACM, 1990.
- [117] STEFFAN, J. and MOWRY, T., “The potential for using thread-level data speculation to facilitate automatic parallelization,” in *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, HPCA ’98, (Washington, DC, USA), pp. 2–, IEEE Computer Society, 1998.
- [118] SULEMAN, M. A., QURESHI, M. K., and PATT, Y. N., “Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on cmps,” in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, (New York, NY, USA), pp. 277–286, ACM, 2008.
- [119] THIES, W., KARCZMAREK, M., and AMARASINGHE, S. P., “Streamit: A language for streaming applications,” in *Proceedings of the 11th International Conference on Compiler Construction*, CC ’02, (London, UK, UK), pp. 179–196, Springer-Verlag, 2002.
- [120] TINYSTM, “<http://www.tmware.org/tinystm>,”
- [121] TINYSTM, “<http://www.tmware.org/tinystm>,”
- [122] TRACHSEL, O. and GROSS, T. R., “Variant-based competitive parallel execution of sequential programs,” in *Proceedings of the 7th ACM international conference on Computing frontiers*, CF ’10, (New York, NY, USA), pp. 197–206, ACM, 2010.

- [123] USUI, T., BEHRENDTS, R., EVANS, J., and SMARAGDAKIS, Y., “Adaptive locks: Combining transactions and locks for efficient concurrency,” pp. 3–14, sept. 2009.
- [124] VANDERWIEL, S. P. and LILJA, D. J., “Data prefetch mechanisms,” *ACM Comput. Surv.*, vol. 32, pp. 174–199, June 2000.
- [125] VOSS, M. J. and EIGENMANN, R., “High-level adaptive program optimization with adapt,” in *ACM SIGPLAN Notices*, pp. 93–102, ACM Press, 2001.
- [126] WINTERSTEIGER, C. M., HAMADI, Y., and MOURA, L., “A concurrent portfolio approach to smt solving,” in *Proceedings of the 21st International Conference on Computer Aided Verification, CAV ’09*, (Berlin, Heidelberg), pp. 715–720, Springer-Verlag, 2009.
- [127] WULF, W. and SHAW, M., “Global variable considered harmful,” *SIGPLAN Not.*, vol. 8, pp. 28–34, Feb. 1973.
- [128] XU, K. and LI, W., “Exact phase transitions in random constraint satisfaction problems,” *Journal of Artificial Intelligence Research*, vol. 12, pp. 93–103, 2000.
- [129] XU, M., BODIK, R., and HILL, M. D., “A ”flight data recorder” for enabling full-system multiprocessor deterministic replay,” in *Proceedings of the 30th Annual International Symposium on Computer Architecture, ISCA ’03*, (New York, NY, USA), pp. 122–135, ACM, 2003.
- [130] YEH, T.-Y. and PATT, Y. N., “A comparison of dynamic branch predictors that use two levels of branch history,” in *Proceedings of the 20th annual international symposium on computer architecture, ISCA ’93*, (New York, NY, USA), pp. 257–266, ACM, 1993.
- [131] YOO, R. M. and LEE, H.-H. S., “Adaptive transaction scheduling for transactional memory systems,” in *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, SPAA ’08*, (New York, NY, USA), pp. 169–178, ACM, 2008.
- [132] YU, J. and NARAYANASAMY, S., “A case for an interleaving constrained shared-memory multi-processor,” in *Proceedings of the 36th Annual International Symposium on Computer Architecture, ISCA ’09*, (New York, NY, USA), pp. 325–336, ACM, 2009.
- [133] ZYULKYAROV, F., HARRIS, T., UNSAL, O. S., CRISTAL, A., and VALERO, M., “Debugging programs that use atomic blocks and transactional memory,” in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP ’10*, (New York, NY, USA), pp. 57–66, ACM, 2010.

- [134] ZYULKYAROV, F., STIPIC, S., HARRIS, T., UNSAL, O. S., CRISTAL, A., HUR, I., and VALERO, M., “Discovering and understanding performance bottlenecks in transactional applications,” in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, (New York, NY, USA), pp. 285–294, ACM, 2010.